(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) Publication number: **0 333 366 A2**

RECEIVED
JUL 28 2003

(12) EUROPEAN PATENT APPLICATION

(21) Application number: 89302226.9

(51) Int. Cl.[4]: G06F 15/06

(22) Date of filing: 06.03.89

(30) Priority: 18.03.88 US 170399

(43) Date of publication of application:
20.09.89 Bulletin 89/38

(84) Designated Contracting States:
DE FR GB IT

(71) Applicant: DIGITAL EQUIPMENT CORPORATION
146 Main Street
Maynard, MA 01754(US)

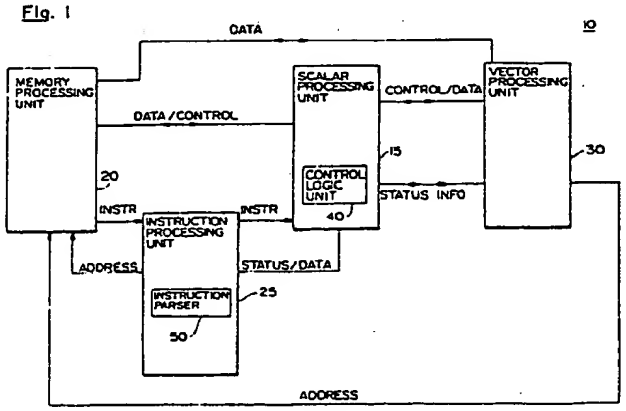
(72) Inventor: Bhandarkar, Dileep P.
3, Lantern Lane,
Shrewsbury Massachusetts 01545(US)
Inventor: Cutler, Dave
3451, Evergreen Point Road
Bellevue WA(US)
Inventor: Cardoza, Wayne
3, Hutchinson Road
Merrimack NH 03054(US)
Inventor: Witek, Rich,
3, Silver Birch
Littleton Massachusetts 01460(US)
Inventor: Orbits, Dave
17028, N.E. 139 Street
Redmond WA 98052(US)

(74) Representative: Goodman, Christopher et al
Eric Potter & Clarkson 14 Oxford Street
Nottingham NG1 5BP(GB)

(54) Context switching method and apparatus for use in a vector processing system.

(57) The invention relates to a method and apparatus to minimize the time a data processing system spends on saving and restoring vector processor state data during a context switch. A context switch occurs when execution of an old process is suspended and execution of a current process is begun. The vector state data associated with the old process remains in the vector processor until the current process attempts to execute a vector instruction. When this occurs, the vector state data associated with the old process is saved and vector state data associated with the current process is restored, if the old process is not the same as the current process. If the old process is the same as the current process, there is no need to save and restore any vector state data.

Fig. 1



EP 0 333 366 A2

BNSDOCID: <EP___0333366A2_I_>

BACKGROUND OF THE INVENTION

This application is related to U.S. Patent Application FINK/P6516, entitled Exception Reporting Mechanism for a Vector Processor, by D. Bhandarkar, et al.; U.S. Patent Application FINK/P6517, entitled Method and Apparatus for Executing Instructions for a Vector Processing System, by D. Bhandarkar et al.; U.S. Application FINK/P6519, entitled, Method and Apparatus for Handling Asynchronous Memory Management Exceptions by a Vector Processor, by F. McKeen et al.

The invention relates to data processing systems with vector processing generally, and specifically to such data processing systems which are capable of executing multiple processes, not all of which may require vector processing.

Certain high performance data processing systems include, in addition to a main or scalar processor, a separate vector processor to process vector instructions quickly and efficiently. Vector instructions direct a processor to perform memory, arithmetic or logical operations on data represented as vectors. The main or "scalar" processor processes the other instructions, often called 'scalar' instructions. Scalar instructions, for example, direct a processor to perform memory, arithmetic or logical operations on logical and scalar data.

Data processing systems that perform multitasking i.e., operate several different tasks or processes) require special handling of vector registers. For example, the IBM 3090 provides multi-tasking and the CPU divides its attention between a plurality of processes. Each process is executed for a short period of time before it is switched out of main memory and another process is brought in. The switching out process is termed a "context switch." Every time a process is "switched out," the current state or context of the machine is saved and the state information of the next process to be switched in is restored.

State information can include such elements as flags (e.g. exception enable flag, etc.), status words (e.g., processor status words, program counters, etc.), scalar registers, and vector registers. All of this information must be stored so that a process that has been "switched out," can, when later "switched in, resume processing exactly where it left off when it was "switched out." The overhead associated with context switching is considerable, especially to store the vector registers. Typical vector processors contain 8 to 16 vector registers with 32 to 128 elements per register. Storage of such registers then requires storing and restoring the contents of 256 to 2048 register elements.

The IBM 3090 uses write flags to avoid switching out every vector register every time a context switch occurs. Whenever a vector register is written to during the execution of a process, a corresponding write flag is set. When the current process is switched out, only the contents of those vector registers that were written to during the execution of the process are saved. This allows the operating system to save only the contents of registers that have been changed since the last save, although at considerable hardware and software expense.

Although this procedure reduces the number of vector registers that need to be saved at every context switch, new register values must be restored for all vector registers before executing the next process. Because no attempt is made to ascertain whether the execution of the next process will require the contents of any of the vector registers, the effort may have been unnecessary.

Thus the saving of an old state and the restoring of a new state at every context switch, even if done only partially as with the IBM 3090, creates significant overhead, especially when a large number of processes are sharing the processor, but only a few processes use vector instructions.

SUMMARY OF THE INVENTION

Accordingly, it is an object of the invention to manage the storage and retrieval of vector registers and flags efficiently in a data processing system executing multiple processes.

Another object of the present invention is to minimize the storage and retrieval of state information for a vector processor.

Additional objects and advantages of the invention will be set forth in part in the description which follows, and in part will be obvious from the description, or may be learned by practice of the invention. The objects and advantages of the invention will be realized and attained by means of the elements and combinations particularly pointed out in the appended claims.

To achieve the objects and in accordance with the purpose of the invention, as embodied and broadly described herein, the data processing system of this invention which sequentially executes a plurality of processes comprises a memory, vector processing means, vector instruction identifying means, last user indicating means, and saving means. The memory has a plurality of portions

BNSDOCID: <EP___0333366A2_I_>

each corresponding to a different one of the processes to be executed by the data processing system. The vector processing means executes vector instructions in the processes and includes vector state means for storing state information describing an execution state of the vector processing means. The vector instruction identifying means recognizes when the current process (i.e., the process currently being executed) is about to execute a vector instruction, and the last user indicating means identifies as the last vector user process the process for which the vector processor most recently executed a vector instruction. The saving means, responsive to the vector instruction identifier and the last user indicator, causes the data processing system to store the vector state information in memory at a location corresponding to the last vector user process when the vector instruction identifier recognizes both that the current process is about to execute a vector instruction and that the last vector user process is different from the current process.

The accompanying drawings, which are incorporated in and constitute a part of this specification, illustrate an embodiment of the invention and, together with the description, serve to explain the principles of the invention.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 is a drawing of a preferred embodiment of a data processing system in accordance with the present invention;

Figure 2 is a block diagram of certain portions of the preferred embodiment of the vector processing unit shown in Figure 1;

Figure 3 is a block diagram of certain portions of the preferred embodiment of the vector control logic shown in Figure 2;

Figure 4 shows various formats for vector instruction words which can be used in a data processing system shown in Figure 1;

Figure 5 shows a vector control word associated with the vector instruction words shown in Figure 4;

Figure 6 shows the FLAGS field of the vector control word shown in Figure 4;

Figure 7 shows one format for a scalar instruction word which can be used in a data processing system according to the present invention;

Figure 8 shows the contents of various registers and vector control word fields during the processing of a vector instruction;

Figure 9 shows a preferred embodiment of an instruction decoder;

Figure 10 is a flowchart of the operations performed when decoding a vector instruction;

Figure 11 is a diagram of a vector processor status register for the vector processing unit shown in Figures 1 and 2;

Figure 12 is a diagram of a vector arithmetic exception register for the vector processing unit shown in Figures 1 and 2;

Figure 13 is a diagram of a vector state address register for the vector processing unit shown in Figures 1 and 2;

Figure 14 is a diagram of a memory management fault stack frame generated by the vector processing unit shown in Figures 1 and 2;

Figure 15 is a flowchart depicting a preferred procedure to be performed by the data processing system shown in Figure 1 when the vector processing unit of that system has experienced a memory management exception;

Figure 16 is a flowchart depicting a preferred procedure to be performed by the data processing system of Figure 1 during context switching;

Figure 17 is a flowchart depicting a preferred procedure to be performed for instruction decoding by the instruction decoder of Figure 9; and

Figure 18 is a flowchart depicting a preferred procedure to be performed by the data processing system in Figure 1 in processing a Vector Processor Disabled Fault set in the procedure shown in Figure 17.

DESCRIPTION OF THE PREFERRED EMBODIMENT

Reference will now be made in detail to a presently preferred embodiment of this invention, an example of which is illustrated in the accompanying drawings. Wherever possible, the same reference numbers will be used throughout the drawings to refer to the same or like parts.

A. General System Description

In accordance with the present invention, there is provided a data processing system, such as system 10, which is capable of executing vector instructions and scalar instructions. Figure 1 shows a data processing system 10 comprising various elements such as a scalar processing unit (SPU) 15, a memory processing unit 20, an instruction processing unit (IPU) 25, and a vector processing unit (VPU) 30.

The data processing system of this invention includes scalar processing means for executing scalar instructions. In the preferred embodiment of

BNSDOCID: <EP 0333366A2 I >

the invention, SPU 15 receives all the instructions from IPU 25, executes the scalar instructions and sends the vector instructions and vector data received from IPU 25 to VPU 30.

In accordance with the present invention, the data processing system also includes vector processing means for executing vector instructions simultaneously with the execution of scalar instructions by scalar processing means. In the embodiment shown in the figures, VPU 30 executes vector instructions simultaneously with the execution of scalar instructions by SPU 15. VPU 30 also contains several vector registers as explained in detail below.

The data processing system of this invention also includes instruction decoding means for routing vector instructions to the vector processing means and scalar instructions to the scalar processing means. As shown in Figure 1, IPU 25 includes an instruction parser 50 which preprocesses instructions received from memory processing unit 20 in a manner described below. Parser 50 sends scalar instructions and data to SPU 15 and sends vector instructions and scalar data to VPU 30 via SPU 15. SPU 15 includes a control logic unit 40 which contains microcode to pass on to VPU 30 the vector instructions and data. Of course, parser 50 could also be designed to send the vector instructions directly to VPU 30. VPU 30 receives vector data from and sends addresses and vector data to memory processing unit 20 without requiring use of IPU 25 or SPU 15.

Memory processing unit 20 receives control, address and data signals from IPU 25, SPU 15 and VPU 30, and then arbitrates, processes and responds to those signals. Data processing system 10 can also include other elements to perform different functions, but an understanding of such elements is not needed for an understanding of the present invention.

Figure 2 is a diagram showing a preferred embodiment of VPU 30. As shown in Figure 2, VPU 30 includes control logic 60 as the main interface with other parts of data processing system 10 and a vector register file 35 for servicing vector data access requests. Such requests can either be write requests composed of write control signals and write addresses, or read requests composed of read control signals and read addresses. Vector register file 35 contains a plurality of write ports, shown as WT PORT0-WT PORT2 and denoted by the reference numbers 41-43, as well as a plurality of read ports, shown as RD PORT0-RD PORT4 and denoted by the reference numbers 51-55. The write ports receive READ/WRITE control signals 45 and write data from vector control logic 60, from a vector multiplier 70, or from a vector adder 80.

The read ports operate similarly to the write ports. For example, read port 53, corresponding to RD PORT0, receives a read enable signal, a vector register select signal, and vector element address signals from control logic 60 via READ/WRITE control signals line 45. The read data for read port 53 is provided to a mask unit 90.

The other read ports also receive their control and address signals from control logic 60. The outputs from read ports 55 and 54, i.e., RD PORT1 and RD PORT2, respectively, are connected to vector multiplier 70, and the outputs of read ports 52 and 51, RD PORT3 and RD PORT4, respectively, are connected to vector adder 80.

Vector register file 35 includes a plurality of vector registers, preferably sixteen, which store vectors processed by VPU 30. Each vector register preferably has sixty-four elements. The size of file 35, however, is not critical to the present invention.

The number of entries of the vector being processed, i.e., the length of that vector is stored in a vector length register (VLR) 182 in control logic 60. In the preferred embodiment, a vector can have up to sixty-four entries and, accordingly, vector length register 182 is 7-bits long to represent vector lengths from 0 to 64 entries.

Vector adder 80 performs integer and floating point add and subtract operations on two vectors supplied from the vector register file 35 via RD PORT3 and RD PORT4. Preferably adder 80 also performs certain logical and shift operations. The output of vector adder 80, labeled "RESULT," provides a data input to WT PORT1. Vector adder 80 also includes exception logic 82 coupled to mask unit 90 which permits adder 80 to perform operations upon condition and to advise mask unit 90 of arithmetic exception conditions.

Vector multiplier 70 performs integer and floating point multiplication and division operations on two vectors received from RD PORT1 and RD PORT2 of vector register file 35. The product or quotient of those inputs is a vector also labeled "RESULT" and is provided as input data to WT PORT2. Exception logic 72, which is coupled to mask unit 90, indicates to mask unit 90 when there is an arithmetic exception condition resulting from the multiplication or division by multiplier 70.

Mask unit 90 receives data from vector register file 35 via RD PORT0 and provides vector data from VPU 30 to SPU 15 via the VECTOR DATA line shown in Figure 2. Mask unit 90 also can read the data from RD PORT0 and convert it to an address for the memory processing unit 20. In addition, mask unit 90 is coupled to exception logic 72 and 82 and latches their exception conditions.

Contained in mask unit 90 is a 64 bit vector mask register 92. Each bit in register 92 corresponds to a different one of the 64 vector ele-

BNSDOCID: <EP___0333366A2_I_>

ments in a given vector register and indicates whether the corresponding vector element is enabled and should be processed. Mask register 92 can be loaded with data from SPU 15, via control logic 60. Preferably, vector mask register 92 ensures that only the results from the enabled elements in a vector register will be stored.

Vector control logic 60 preferably includes an instruction parser 65 and vector save and decode logic 66 to implement certain control functions. Vector save and decode logic 66 in control logic 60 oversees the scheduling of all activity in VPU 30 and performs certain data transfer activity.

Instruction parser 65, portions of which are shown in greater detail in Figure 3, receives information (i.e., instructions via the VECTOR DATA line) from $\overline{\text{SPU}}$ 15 and directs incoming data and addresses to the appropriate read and write ports and then to a vector multiplier 70, vector adder 80, or mask unit 90. Preferably, instruction parser 65 passes scalar data received from SPU 15 on a SCALAR DATA line to be stored, preferably, in scalar registers (not shown) in vector register file 35 for use during vector processing operations.

Figure 3 shows the internal logic of instruction decoder 65 in VPU 30. Decoder 65 receives from SPU 15, as inputs 100, partially decoded vector instruction information and scalar data. These inputs are stored in instruction buffer 115 which holds the instructions and the data in the proper order until the vector instructions can be executed by VPU 30.

Multiplexer 116, which is connected to the outputs BUF0-BUF3 of buffer 115, then correctly outputs the instruction information for the next instruction to be decoded. That information is determined by the instruction select signal which is a two-bit recirculating counting signal that makes buffer 115 appear to operate as a circulating buffer.

Scalar data is selected by multiplexer 102, which is also connected to receive the BUF0-BUF3 outputs from buffer 115, according to the Instruction Select + 1 signal, which has a value of 1 larger than the Instruction Select signal. The scalar data is driven to vector register file 35, where it is stored in a scalar register corresponding to either vector adder 80, vector multiplier 70, or mask unit 90.

If the vector instruction being decoded requires only vector operands, then the information is passed through buffer 116 and the operands are determined according to control word decoder element 103 in a manner described below. If an instruction requires scalar data, however, that data is passed from SPU 15 following the instruction and is output at the correct time by multiplexer 102.

After the instruction is selected by multiplexer 116, certain fields are distributed to decode logic elements 103, 104, 105 and 106. The instruction information in the partially decoded instruction includes an opcode portion indicating the type of instruction, a control word type portion indicating the type of the control word (described below in greater detail), a dispatch type portion indicating which of the main vector units, such as adder 80 or multiplier 70, is to be used by the instruction, and a control word portion specifying the operands and flags for the vector instruction.

Control word decoder element 103 strips the operands from the control word portion to form the addresses for the registers in the vector register file 35 both for the vector registers which provide the source of data for the specified vector operations, i.e., SOURCE REG1 and SOURCE REG2, and the vector register which stores the result of the vector operation, i.e., DEST REG. In addition, the decoder element 103 also strips out the Exceptions/Mask flags.

The values for SOURCE REG 1, SOURCE REG 2, and DEST REG are inputs to address generators 108, 109, and 110, respectively, which generate addresses for the vector adder 80, the vector multiplier 70, and mask unit 90, respectively, according to the type of instructions. Preferably, address generators 108, 109 and 110 are counters which determine the particular suboperation being performed and are shown as counters 44 in vector control logic 60, as shown in Figure 2.

The other instruction type portions are sent to dispatch type logic 104, control word type logic 105, or the opcode decode logic 106 for further decoding. Based on such decoding, issue decision logic 107 then determines whether that instruction can issue. For example, if vector adder 80 is busy, then the next instruction must be delayed if it requires adder 80 until adder 80 is free. When the issue decision logic 107 determines that the next instruction can execute, it issues an Issue New signal which causes address generators 108, 109, and 110 to load their proper value.

B. Vector Instruction Format

Figure 4 shows the formats of various vector instruction words having different numbers of operand specifiers. Figure 5 shows different formats of a vector control word pointed to by a vector control word specifier 140 of a vector instruction word having the format shown in lines (a)-(d) of Figure 4.

As shown in Figure 4 of the present embodiment, vector instructions contain opcode field 150 which holds an operation code identifying the type of instruction, and an operand pointer section 152.

BNSDOCID: <EP 0333366A2 I >

Operand pointer section 152 may contain any number of operand specifiers 154-158 and a vector control word specifier 140. Preferably, all operand specifiers in the operand pointer section 152 have the same format as the operand specifiers in a scalar instruction, and, like the specifiers in the scalar instruction, the operand specifiers in the vector instruction words identify the location of a scalar operand. Because the vector operand specifiers and scalar operand specifiers (see Figure 7) have identical formats, operands may be fetched before knowing whether a given instruction is a vector instruction or a scalar instruction.

Figure 7 shows a scalar instruction word having three operand specifiers 154', 155', and 156'. Vector control word specifier 140 in lines (a) through (d) of Figure 4 and vector operand specifiers 154', 155', and 156' of Figure 7 use identical address schemes. thus freeing instruction parser 50 from having to fetch operands differently for vector instructions and scalar instructions. Use of an identical address scheme allows the organization of VPU 30 to be transparent to IPU 25. The preferred addressing scheme by which operand specifiers 154-158 identify operands is explained in United States Patent No. 4,241,399 issued to Strecker et al., which is herein incorporated by reference.

Unlike the operand specifiers 154-158, vector control word specifier 140 points to a 16 bit vector control word 160, three examples of which are shown in Figure 5. Vector control word 160, as shown in Figure 5, contains 16 bits, but could be any length, such as 8, 32 or 64 bits. In Figure 5(a), vector control word 160 is divided into four 4-bit fields 162-165. Preferably, field 162 is a flags field and either one, two or three of the fields 163-165 are vector register specifier fields which contain pointers to, or are specifiers of, vector register operands to be accessed during execution of vector instructions. Each vector register specifier preferably identifies a unique vector register. The four bits of each vector register specifier field can uniquely identify one of the sixteen vector registers contained in the current embodiment of the invention.

Figure 5(b) shows an alternate format for a vector control word 160'. In addition to flags field 162 and vector specifier fields 163 and 164, control word 160' includes a 4-bit operand information field 168 instead of the 4-bit vector register specifier field 165 of Figure 5(a). Operand information field 168 contains information required for various vector operations. For instance, if the operation code in opcode field 150 indicates that a compare operation is to be performed, the operand information field 168 may contain data specifying whether a "greater than" or "less than" comparison is to be performed. The operand information field 168 may also occur in place of vector register specifier 163, vector register specifier 164, or vector register specifier 165, depending on the value of the opcode field 150.

Figure 5(c) shows yet another format for a vector control word 160". Control word 160" contains a convert field 166 in place of the vector specifier field 163. Convert field 166 identifies a conversion function to be performed so that values in the vector register specified by field 164 are properly converted before storage in the vector register specified in field 165. Such conversion can include, for example, a floating point to integer or long word conversion, etc.

Figure 6 shows an expansion of the flags field 162 of the vector control word 160 of Figure 5(a). One bit contains a mask operation enable (MOE) flag 170. A second bit contains a match true/false (MTF) flag 172. MTF flag 172 determines whether a Boolean value of zero or one represents an enabled bit in vector mask register 92. A third bit contains the exception enable (EXC) flag 174 which enables exception handling when vector processing exceptions occur. A fourth flag bit 176 is presently unused. The use of these bits is dependent on the instruction being processed at the time.

As explained briefly above, in the masking function enabled by MOE flag 170, only those results of a vector operation corresponding to an enabled bit in vector mask register 92 are stored after an operation. Figure 8 shows an example of how the vector mask register 92, the vector length register 182, and the MOE and MTF flags 170 and 172, respectively, of the flags field of vector control word 160 affect the storage of a result of a vector operation. In Figure 8(a), the MOE flag 170 of the vector control word contains a value of "1" indicating that masking is enabled. The MTF flag 172 contains a value of "0," indicating that only those results corresponding to values of "0" in the vector mask register 92 are to be stored.

Figure 8(b) shows the contents of a destination register 192 of vector register file 35 before a vector operation has occurred, that is, before result data from a vector operation has been stored in the destination register 192. Figure 8(c) shows vector mask register 92 in which bits set to "0" are enabled bits because of the MTF field 172 in Figure 8(a).

Figure 8(d) shows a result 190 of a vector operation before masking and before the vector result 190 has been stored in destination register 192 (the values in result 190 have been chosen arbitrarily for the sake of example). Note that, in Figure 8b, the two leftmost elements of the destination register 192 contain values of "9" and "7," respectively. After the vector operation, the two leftmost elements of the vector result 190 contain

BNSDOCID: <EP 0333366A2 I >

values of "4" and "5," respectively, as shown in Figure 8(d). The leftmost element of the vector result 190 corresponds to a bit in the vector mask register 92 having a value of "0." The second leftmost bit in the vector result 190 corresponds to a bit in vector mask register 92 having a value of "1."

Figure 8(e) shows destination register 192 after masking, when certain elements of vector result 190 of Figure 8(d) have been stored in destination register 192. The leftmost element of destination register 192 contains the value "4." This value was stored into destination register 192 because a corresponding bit in the vector mask register 92 contained a value of "0." The second leftmost element of destination register 192, however, still retains its original value of "7" in Figure 8(e). The result value of "5" in vector result 190 in Figure 8(d) was not stored into destination register 192 in Figure 8(e) because a corresponding bit in the vector mask register 92 contained a "1." Similarly, all other elements of vector result 190 which correspond to enabled bits of the vector mask register 92 are stored in destination register 192, but no element of the vector result 190 corresponding to a disabled vector mask register bit is stored in destination register 192.

C. Instruction Preprocessing

Prior to SPU 15's routing of instructions either to VPU 30 or to other circuitry in SPU 15 for execution, IPU 25 performs certain preprocessing on the instructions. As explained briefly above, the unique nature of the instruction formats discussed above allows IPU 25 to use the same circuitry, a preferred embodiment of which is shown in Figure 9 as including buffer 187 and operand control logic 185, to preprocess both vector instructions and scalar instructions identically.

Figure 10 is a flow chart of a preferred procedure for preprocessing instructions in IPU 25. When memory processing unit 20 transmits an instruction to IPU 25, that instruction is input to buffer 187 (step 194 in Figure 10). The operation code portion of that instruction (i.e., opcode field 150 shown in Figures 4(a)-(d)) is then stripped away (step 196), and transmitted to SPU 15. The operand pointer portion of that instruction (i.e., operand pointer section 152 in Figures 4(a)-(d)) becomes an input to operand control logic 185, which also has access to the operation code portion if needed to perform its functions.

Operand control logic 185 decodes the operand specifier portions, using the operation codes if necessary, and performs any accesses to memory processing unit 20 necessary to determine the operands pointed to by each operand specifier portion (step 198). The details of the procedure to determine operands are explained in the Strecker et al. patent referred to above. Finally, IPU 25 sends the operands and the operation code portions to SPU 15 either for execution by SPU 15 or for transmission to VPU 30 of vector instructions and operands (step 199).

Because the vector and scalar processing instructions have the same format, operand control logic 185 performs the same function for both vector and scalar instructions. Indeed, for any type of data processing means that has a separately operating processor, such as an FFT processor or a floating point processor, the use of the instruction formats described above allows a single instruction processor, such as IPU 25 to preprocess all instructions in the same manner.

D. Vector Processing Exceptions

1. Vector Processor Registers

In accordance with the present invention, the vector processing means includes state means for holding vector state information representing an execution state of the vector processing means. In the preferred embodiment of this invention, a vector processor status register (VPSR) 200, shown in Figure 11, is provided in VPU 30 to hold information representing VPU 30's execution state. VPSR 200 holds 32 bits of status information about VPU 30 and is preferably located in control unit 60 although it could be located elsewhere. Also located in mask unit 90 are a vector arithmetic exception register (VAER) 300, shown in Figure 12, and a vector state address register (VSAR) 400, shown in Figure 13. The contents of registers 200, 300 and 400, as well as other data discussed below, are part of the state information of VPU 30.

Bit zero of VPSR 200 is a vector processor enable (VEN) bit 210 which indicates whether VPU 30 is disabled. VPU 30 is enabled by writing a 1 to this bit and disabled by writing a 0 to this bit. When VPU 30 is disabled, any attempt by SPU 15 to send vector instructions to VPU 30 results in a Vector Processor Disabled Fault.

Bit one of VPSR 200 is a vector processor state reset (RST) bit 220. Writing a one to this bit clears VPSR 200, except for VEN bit 210.

Bit two of VPSR 200 is a vector state store (STS) bit 230 which, if set to a one, initiates the storing of implementation specific vector state information to memory processing unit 20 using a virtual memory address 420 in VSAR 400 for asynchronous handling of memory management excep-

BNSDOCID: <EP 0333366A2 >

tions. If synchronous handling of exceptions is implemented, STS bit 230 is ignored. The preferred embodiment of the invention allows the selection of one of two ways of executing vector processing memory access instructions. The two ways result in two different mechanisms for handling memory management exceptions, as explained in the next section. One is synchronous memory management execution and the other is asynchronous memory management execution. During synchronous execution, VPU 30 and SPU 15 process no new instructions until the vector memory access instruction currently executing is guaranteed to complete.

For asynchronous execution, VPU 30 can execute a vector memory access instruction simultaneously either with SPU instruction execution or with VPU execution of other vector memory instructions. Asynchronous memory management allows chained execution of an instruction sequence such as LOAD/ADD/MULTIPLY/STORE.

Bit three of VPSR 200 is a vector state reload (RLD) bit 240 which, if set to a one, initiates reloading of implementation-specific vector state information from memory using virtual memory address 420 in VSAR 400 for asynchronous handling of memory management exceptions. As with STS bit 230, if synchronous handling of exceptions is implemented, RLD bit 240 is ignored.

Bit five of VPSR 200 is a memory fault (MF) bit 250 which is set to a one by VPU 30 to indicate the presence of a memory reference to be re-executed due to an asynchronous memory management exception. This is explained in greater detail below. If synchronous handling of memory management exceptions is implemented, this bit is set to zero.

Bit six of VPSR 200 is a pending memory fault (PMF) bit 260. VPU 30 sets PMF bit 260 to a one to indicate that an asynchronous memory management exception is pending. If synchronous handling of memory management exceptions is implemented, this bit is always zero.

Bit seven of VPSR 200 is an arithmetic exception (AEX) bit 270 which is normally zero to indicate that VPU 30 is not disabled by certain arithmetic exceptions, such as a floating underflow or integer overflow. VPU 30 always sets this bit when an arithmetic exception occurs. Information regarding the specific nature of the exception is located in VAER 300.

Bit twenty-four of VPSR 200 is an implementation-specific hardware error (IMP) bit 280. IMP bit 280 is set to one when VPU 30 is disabled due to a hardware error.

Bit thirty-one of VPSR 200 is a vector processor busy (BSY) bit 290 which VPU 30 sets to one when it is executing vector instructions. When this bit is cleared to zero, the VPU 30 is idle or has suspended instruction execution due to an asynchronous memory management exception.

Preferably VAER 300 is a register used to record information regarding vector arithmetic exceptions. VAER 300 is located in mask unit 90 and is written to by exception units 82 and 72. Control logic unit 40 of the SPU 15 can read the contents of VAER 300 but cannot write to VAER 300. VAER 300 includes two fields in the preferred embodiment: a vector destination register mask 310 and an exception condition summary 320. Vector destination register mask field 310 of VAER 300 records which vector registers have received a default value due to arithmetic exceptions. If the nth one of the sixteen vector registers receives a default value as a result of an arithmetic exception, a one is written to bit n of mask field 310 (bit (16 + n) of VAER 300).

Exception condition summary field 320 indicates the type of exception which has occurred. Preferably those conditions include floating underflow, floating divide by zero, floating reserved operand, floating overflow or integer overflow exception conditions.

As explained above, instruction parser 50 in IPU 25 passes vector and scalar instructions to SPU 15. SPU 15 then identifies vector instructions and passes them to VPU 30. Of course, the vector instructions could as well be passed directly to VPU 30 from IPU 25. VPU 30 then executes the vector instructions in accordance with its normal operation. During execution of the vector instructions, however, two types of vector processor exceptions may occur: vector memory management exceptions and vector arithmetic exceptions.

2. Memory Management Exceptions

The two types of vector processor exceptions are handled slightly differently. Unlike vector arithmetic exceptions, vector memory management exceptions do not disable VPU 30. Memory management exceptions which occur during a memory access instruction prevent further processing of that instruction, which is not the case with arithmetic exceptions. Memory management exceptions will also halt the execution of all other instructions which depend on the data.

The vector processing means of this invention includes exception detecting means for indicating the presence of a memory management exception. In the preferred embodiment of VPU 30, control logic 66 is notified by memory processing unit 20 when a memory management exception occurs. The details of this procedure are explained below.

Vector memory management exceptions include access control violations, translation not valid exceptions, modify exceptions and vector align-

BNSDOCID: <EP___0333366A2_I_>

ment exceptions. Preferably, memory processing unit 20 provides signals identifying the type of memory management exception, although such signals could as well be in SPU 15 or VPU 30. These exception identifying signals are part of the state information of VPU 30.

Access control violations involve attempts by VPU 30 to access protected portions of memory or portions of memory outside of the memory space allocated to the current vector process. The current vector process refers to the process containing the vector instruction whose execution caused the memory management exception. A translation not valid exception occurs when there is a page fault or some other attempt to access a portion of memory not currently in memory processing unit 20. A modify exception occurs when the access to a page in memory involves a modification to a page being made for the first time by the current vector process. Vector alignment exceptions occur when long words (which occupy four bytes) or quadwords (which occupy eight bytes) do not fall in long word or quadword boundaries, respectively, in memory processing unit 20. A vector alignment exception is also considered an access control violation because similar software handles both conditions.

If more than one kind of memory management exception occurs during the processing of a single vector instruction, then a certain hierarchy of exception handling must be followed. The highest priority goes to access control exceptions (and accordingly to vector alignment exceptions which are treated similarly) because they are the most serious. Such exceptions cause data processing system 10 to remove the current vector process from execution. The translation not valid exception has the next highest priority.

The vector processing means of the present invention also includes vector halt means for stopping execution of the faulting vector processing instruction by the vector processing means when a memory management exception occurs and for allowing the scalar processing means to continue executing scalar instructions. When a memory management exception occurs, vector state and decode logic 66 causes VPU 30 to discontinue executing vector instructions. This is done by stopping the transmission of valid data to VPU 30.

In accordance with the present invention, the data processing system includes exception record means for recording an indication of the occurrence of a memory management exception and sufficient information about the state of the vector processing means so that the vector processing means can later resume execution at the suboperation of the vector processing instruction during which the memory management exception occurred. In the preferred embodiment of this invention, during both synchronous memory management and asynchronous memory management, the occurrence of a memory management exception causes SPU 15, and in particular control logic unit 40 in SPU 15, to create a memory management stack frame which it stores at a predetermined location in memory processing unit 20. The software for handling the memory management exception is designed to recognize the predetermined location containing the memory management stack frame so appropriate processing, which can include well-known routines appropriate to the specific exception, can proceed expeditiously.

Figure 14 illustrates an example of a memory management stack frame 500. Stack frame 500 contains a memory management fault code 510, an address word 520, a program counter PC 530 and a program status word PSL 540.

Fault code 510 preferably contains a length bit 511, a page table reference bit 512, a modify bit 513, a vector alignment bit 514, a vector I/O bit 515, and a vector asynchronous memory management exception bit 516. Length bit 511 is set when there is an access control violation caused by an access to a location in memory processing unit 20 outside of the memory space allocated to the vector process using VPU 30. Page table reference bit 512 is set if a memory management exception occurred while accessing a page table. Modify bit 513 is set if the instruction causing the exception is a write to memory processing unit 20. Vector alignment (VAL) bit 514 is set when the exception is due to the misalignment of a vector element. Vector I/O (VIO) bit 515 is set when an access control violation occurs during a vector instruction reference to an address in I/O space. Vector asynchronous memory management (VAS) bit 516 is set when a vector processor memory management exception occurs while asynchronous memory management is implemented.

During synchronous memory management, exceptions are handled immediately and execution of the faulting instruction is then restarted. The instruction is restarted by backing the vector memory access instruction up to the beginning. At that point, a memory management fault is initiated and the value in PC 530 identifies the faulting vector memory access instruction. VSAR 400 is ignored during synchronous memory management exceptions because memory management exceptions are handled when they occur, so there is no need to store VPU 30 state information.

Address 520 in stack frame 500 represents a virtual address in the page in memory processing unit 20 whre the exception occurred. PC 530 is the program counter of the instruction which triggered the exception. Note that the instruction triggering

the exception is different from the instruction performing the vector memory access in the case of asynchronous memory management. PSL 540 is a program status word for SPU 15.

As explained above, during asynchronous memory management, not only is SPU 15 allowed to execute scalar instructions simultaneously with VPU 30, but VPU 30 can be executing vector arithmetic instructions along with a vector memory access instruction. If a memory management exception occurs in implementations using asynchronous memory management, then VPU 30 takes steps to freeze its state so that state can be stored and later recalled in order to resume processing at the point where the exception occurred.

Specifically, when VPU 30 executes a vector memory access instruction, decode and control logic 66 actually executes several suboperations necessary to implement such an instruction. For example, for a vector load operation, decode and control logic 66 executes, for each vector element from 0th element to the nth element (n = the contents of VLR 82 -1), the following suboperations:

1. implement any mask functions;
2. check vector alignment; and
3. load the corresponding vector element with the contents at the next location of memory processing unit 20, beginning with an address generated from an operand of the vector memory access instruction.

Associated with each vector register involved in executing a vector instruction is one of several counters 44 in vector register control logic 60 which counts down from n to 0 as the suboperations are being performed, and which provides an address for the associated vector elements in the associated vector register. If a vector instruction is stopped due to a memory management exception, the value of one of the counters 44 associated with the faulting instruction indicates the suboperation at which the exception occurred. That value, as well as the counter associated with any other unfinished instruction, is preferably stored as part of VPU 30 state information. When VPU 30 is reloaded with the stored state information, and the memory management fault is called, counters 44 are reloaded with the suboperation values so the execution of the faulting instruction and all unfinished instructions can begin where the exception occurred. This hardware is shown in greater detail in U.S.S.N. 093,499, which is herein incorporated by reference.

If, during the processing of a vector memory access instruction, memory processing unit 20 notifies vector decode and control logic 66 of a memory management exception, logic 66 then sets PMF bit 260 and MF bit 250 of VPSR 200. VPU 30, however, does not inform SPU 15 of the exception condition, thus allowing SPU 15 to continue processing scalar instructions.

In addition, logic 66 allows VPU 30 to finish executing any vector instructions that had started when the exception occurred, so long as those vector instructions do not use source data whose validity might be affected by the exception condition. This is done by examining the source and destination registers. As explained above with regard to Figure 3, address generators 108, 109, 110 identify the vector registers in vector register file 35 containing the data for the vector instruction. These same address generators also identify the vector register receiving the results of the vector instructions. If logic 66 determines that the DEST REG values for a faulting instruction are the same as the SOURCE REG values for another instruction, that other instruction is also halted.

The memory management exception is not handled further until the next time SPU 15 attempts to send a vector instruction to VPU 30. At that time, SPU 15 begins executing a procedure to send the vector instruction to VPU 30, the preferred method of which is shown by the flowchart in Figure 15.

Before sending a vector instruction to VPU 30, SPU 15 first checks VEN bit 210 to see whether VPU 30 is disabled (step 700). SPU 15 thus acts as an instruction blocking means according to the present invention for preventing the routing of vector instructions to VPU 30 when VEN bit 210 indicates VPU 30 is disabled. If so, SPU 15 concludes that something other than a memory management exception has occurred (at least in the preferred embodiment), and takes a trap for a Vector Processor Disabled Fault (710). The routine for handling this fault is discussed in the following description of arithmetic handling exceptions. Means for sensing when VPU 30 is disabled and for setting the trap are contained in different circuitry or portions of microcode in control logic unit 40 of SPU 15.

If VPU 30 is not disabled, SPU 15 then checks PMF bit 260 (step 720). If PMF bit 260 is set, there has been an unreported memory management exception. SPU 15 resets that bit (step 730) and enters a memory management fault handler to process the vector processor memory management exception (step 735). That fault handler is a standard memory management fault handler well-known to persons of ordinary skill.

If PMF 260 was not set (step 720), SPU 15 next checks MF bit 250 (step 740). If MF bit 250 is set, indicating that the cause for the memory exception has been corrected but the suboperation causing the exception has not yet been executed, SPU 15 clears the MF bit (step 743), and restarts VPU 30 to retry the faulting memory reference

again (step 746).

After VPU 30 is restarted, or if the MF bit 250 was not set, a VPU 30 instruction queue (not shown) is checked to see whether it is full (step 750). If so, then the procedure in Figure 15 is reentered. If the VPU 30 queue is not full, then SPU 15 issues the next instruction (step 760), which is the pending vector instruction that initially triggered the memory management exception.

3. Arithmetic Exceptions

Unlike the situation with vector memory management exceptions, vector instructions which encounter vector arithmetic exceptions are always executed to completion. If an arithmetic exception occurs, either a default or truncated result is written into the corresponding vector register element. The arithmetic exception condition type and destination register number are recorded in exception condition summary field 320 of VAER 300. Mask unit 90 in VPU 30 then reads EXC bit 174 for the vector instruction to determine whether exceptions are enabled. As explained above, this bit enables floating underflow and integer overflow exceptions.

If EXC bit 174 is set for an instruction causing a floating underflow or integer overflow, VPU 30 sends a default reserved operand value for the floating point underflow or the low order bits for an integer overflow to the appropriate destination register element, clears VEN bit 210 of VPSR 200, and stores the appropriate arithmetic exception information in VAER 300. VPU 30 continues processing. VPU 30 follows the same procedure for floating overflow, floating divide by zero, and floating reserved operand arithmetic exceptions, which cannot be disabled.

The sixteen bits of the vector destination register mask 310 of VAER 300 indicate which of the sixteen vector registers have received a default value due to the occurrence of arithmetic exceptions. The types arithmetic exceptions include floating underflow, floating-divide-by-zero, floating reserved operand, floating overflow and integer overflow. When a vector arithmetic exception occurs, VPU 30 disables itself. VPU 30 will complete the execution of the vector instruction causing the exception as well as any other instructions already in VPU 30, but will then refuse to accept subsequent vector instructions from SPU 15 until reenabled. VPU 30 is disabled by writing a zero to VEN bit 210 in VPSR 200.

If SPU 15 later attempts to send a vector instruction to VPU 30 when it is disabled, a Vector Processor Disabled Fault occurs.

4. Context Switching and Instruction Decoding

Memory management exceptions and arithmetic exceptions are most difficult to handle if VPU 30 and SPU 15 are executing instructions from alternately executing processes or programs. This condition occurs whenever there has been a context switch. In general, a great deal of system "overhead" is expended during context switching, and even slight savings may make a data processing system much more efficient.

Figure 16 shows a flowchart of a preferred algorithm performed during a context switch as execution of one process, called the "last process," is halted and execution of a new process, called the "current process," is begun. Once a context switch occurs (step 800), SPU 15 waits until VPU 30 is not busy (step 801). A not busy state is indicated by BSY bit 290 of VPSR 200 having a value of zero. Once VPU 30 is not busy, SPU 15 saves only the scalar state information of the last process (step 802). That state information is saved by storing it into locations in memory processing unit 20 which correspond to the last process. SPU 15 contains its own process status register (see PSL 540 in Figure 14) whose contents are saved as part of the step of saving the last SPU 15 state. During this step, the contents of all the registers in SPU 15 are also saved.

The state information of VPU 30 is not stored at this time. This delays, and may eliminate, the overhead caused when the state information of VPU 30 is stored. Storing the state information of VPU 30 would not only require storage of VPSR 200, VAER 300, VSAR 400, and the other system registers, but, more significantly, it would require the storage of the 16 vector registers, each with 64 elements, which would occupy a great deal of system resources.

Next, SPU 15 disables VPU 30 by clearing VEN bit 210 in VPSR 200 (step 804). Clearing VEN bit 210 prevents the current process from executing a vector instruction without incurring a Vector Processor Disabled Fault. In fact, as is explained in greater detail below, the state of VPU 30 for the last process need not be stored during execution of the current process if the current process does not execute a vector instruction.

Next, the scalar state information of the current process is loaded from locations in memory processing unit 20 corresponding to the current process (step 806), and the current process then begins execution (step 808). Because the vector state of the current process is not loaded during a context switch, VPU 30 contains the state information of the last process to execute a vector instruction. However, this state is inaccessible to the current process because VPU 30 is disabled.

BNSDOCID: <EP___0333366A2_I_>

Figure 17 shows a flowchart of a preferred algorithm to be performed each time a process begins executing a new instruction. Preferably the steps shown in Figure 17 are performed in SPU 15.

If the instruction is a privileged instruction (step 900) and SPU 15 is not in the privileged mode (step 902), an illegal instruction trap occurs (step 904) and the algorithm shown in Figure 17 is completed. The procedure for handling the illegal instruction trap is common to almost every data processing system containing privileged instructions, and is well within the knowledge of a person of ordinary skill.

If the instruction is a privileged instruction (step 900) and SPU 15 is in the privileged mode (step 902), than the instruction is executed (step 905).

If the instruction is not a privileged instruction (step 900) and is not a vector instruction (step 908), SPU 15 executes the instruction (step. 910) If, however, the instruction is a vector instruction (step 908), microcode in control logic unit 40 tests VEN bit 210 of VPSR 200 (step 912). If VEN bit 210 is set, indicating that VPU 30 is enabled, VPU 30 executes the instruction (step 914).

If VEN bit 210 is not set (step 912), then a Vector Processor Disabled Fault occurs (step 916). As explained above, VEN bit 210 is not set when the current process has not yet executed a vector instruction or if an arithmetic exception condition occurs. If the current process attempts to execute a vector instruction when VEN bit 210 is not set, the system must ensure that the vector state of the current process is switched into memory and that any exception conditions are processed before the vector instruction of the current process is executed.

Figure 18 shows a flowchart of a preferred algorithm to process a Vector Processor Disabled Fault, such as that shown in step 916 of Figure 17. The purpose of this algorithm is three-fold: (1) to process any vector exception conditions; (2) to store all VPU 30 state information from the process which last executed a vector instruction; and (3) to retrieve the VPU 30 state information for the current process. Items (2) and (3) are not necessary if the current process is the same as the process which last used VPU 30. Item (1) is necessary only when an exception condition is pending. It is possible, however, that items (1), (2), and (3) must all be completed before the currently executing process executes a vector instruction.

The first step in the Vector Processor Disabled Fault operation is to determine whether the current process is also the last process to have used VPU 30 to execute a vector instruction (step 1000) by comparing the process identification numbers stored in the memory locations designated for the current process and last process identifiers. Alternatively, this operation can also be effected by special hardware registers provided in SPU 15. If the determination in step 1000 is "YES," it is unnecessary to update the state information for VPU 30. If the determination in step 1000 is "YES," then it is determined whether there are any pending arithmetic or memory management exception conditions (step 1002). As explained above, unlike prior art systems, data processing system 10 does not process the exceptions when they occur, but instead processes them when VPU 30 is used next. This may be the same time that VPU 30 saves its vector state information and retrieves the vector state information for the current process. The presence of exception conditions is determined by SPU 15's examination of PMF bit 260 (memory management exceptions) and AEX bit 270 (arithmetic exceptions) of VPSR 200.

If there is a pending arithmetic exception, then AEX bit 270 is cleared (step 1003) and the exception is processed (step 1004) in a manner appropriate to the exception in accordance with a procedure known to persons of ordinary skill in the data processing field. If there is no pending arithmetic exception condition (step 1002), it may be necessary to determine whether a context switch has occurred since the last Vector Processor Disabled Fault. Even when the current process is the same as the process which last used VPU 30 (step 1000) and, therefore, there is no need to save or restore VPU 30 vector state information, it may be necessary to update registers in VPU 30 which are affected by normal scalar processing. If, for example, VPU 30 contains copies of memory management registers located in SPU 15, the registers in VPU 30 would have to be updated after every context switch to ensure that their contents match the contents of the SPU 15 registers.

Next, VPU 30 is enabled by setting VEN bit 210 (step 1010). Control then returns to step B of the instruction decoding algorithm of Figure 17 (step 1012) and the current process begins executing a new instruction.

If the current process is not the same as the process which last used VPU 30 (step 1000), then several steps occur. SPU 15 first waits until VPU 30 is not executing a vector instruction (step 1014). This not busy state is indicated by a zero value in BSY bit 290 of VPSR 200. Second, it is determined if any pending arithmetic exceptions exist (step 1016). If AEX bit 270 of VPSR 200 is set, indicating the presence of a pending arithmetic exception, the arithmetic exception state, as contained in VAER 300, is saved and the existence of an arithmetic exception for the last process to use VPU 30 is noted for later processing (step 1018). Notation of such an exception preferably is accomplished via a software flag associated with the last process to

use VPU 30.

The next step is to determine if a pending asynchronous memory management exception exists (step 1020). Preferably, if such an exception exists, as indicated by a value of one in bit MF 250 of VPSR 200, a software flag associated with the last process to use VPU 30 is set and implementation specific vector information is saved for that process (step 1022).

Next, VPU 30 is enabled by setting VEN bit 210 of VPSR 200 and VPU 30 is reset by setting RST bit 200 of VPSR 300, (step 1024). Since the current process is not the same as the process which last used VPU 30 (step 1000), then the state information of VPU 30 must be updated. This is done in two steps. First, the state information in VPU 30 is stored into an area of memory processing unit 20 corresponding to the process which last used VPU 30 (step 1026). There may also be provided memory management registers in SPU 15 and corresponding memory management registers in VPU 30 to effect virtual addressing and memory protection functions. In this case, the memory management registers in VPU 30 may need to be updated to ensure that they match the corresponding memory management registers in SPU 15. Next the state information for VPU 30 for the current process is retrieved from an area of memory processing unit 20 corresponding to the current process and is stored into the vector registers of VPU 30 (step 1027), and the identification number of the current process is stored in the memory location or special hardware register designated for identifying the last vector process.

Next, a software flag associated with the current process is checked to determine if there is a pending asynchronous memory management exception for the current process (step 1028). If such a pending exception exists, the software flag for the current process is cleared and the vector state at the time of the asynchronous memory management exception is restored (step 1030). This is accomplished by writing the address of the saved vector state information to VSAR 400 and by setting RLD bit 240 and VEN bit 210 of VPSR 300, as discussed above in the section describing the registers of VPU 30.

Next a software flag associated with the current process is checked to determine if there is a pending arithmetic exception for the current process (step 1032). If such a pending exception exists, bit VEN 210 of VPSR 300 is cleared, disabling VPU 30, and the arithmetic exception is processed, similarly to step 1004, in a manner in accordance with a procedure known to persons of ordinary skill in the data processing field (step 1034).

When the Vector Processor Disabled Fault handling algorithm of Figure 18 is complete (step 1036), control returns to step B of the instruction decoding algorithm of Figure 17 and the current process begins executing a new instruction.

Context switching, which for vector processors can be a costly operation, is thus simplified significantly. The overhead accompanying such switching is delayed until it may no longer be necessary.

It will be apparent to those skilled in the art that various modifications and variations can be made in the data processing system and methods of the present invention and in the construction of this system without departing from the scope or spirit of the invention. Other embodiments of the invention will be apparent to those skilled in the art from consideration of the specification and practice of the invention. It is intended that the specification and examples be considered as exemplary only, with a true scope and spirit of the invention being indicated by the following claims and their equivalents.

Claims

1. A multitasking data processing system for alternately executing portions of a plurality of processes in succession, said processes consisting of sequences of instructions, the one of said plurality of processes currently being executed by said data processing system being called a current process, the data processing system comprising:

a memory having a plurality of portions, each corresponding to said processes;

scalar processing means for executing scalar instructions;

vector processing means for executing vector instructions in said processes, said vector processing means including vector state means for storing state information describing an execution state of said vector processing means;

instruction decoding means for identifying and routing scalar instructions and vector instructions to said scalar processing means and said vector processing means, respectively;

vector instruction identifying means for recognizing when said current process is about to execute one of said vector instructions;

last user indicating means for identifying as a last vector user process the one of said processes for which said vector processing means most recently executed a vector instruction; and

saving means, responsive to said vector instruction identifying means and said last user indicating means, for causing said data processing system to store said vector state information in said memory at one of the locations corresponding to the last vector using process when said vector instruction identifying means recognizes both that said current

BNSDOCID: <EP___0333366A2_I_>

process is about to execute a vector instruction and that said last vector user process is different from said current process.

2. The data processing system of claim 1, further comprising:
restoring means, responsive to said saving means, for causing said data processing system to store into said vector state means the contents from a portion of said memory corresponding to said current process.

3. The data processing system of claim 1, further comprising:
context switching means for recognizing when said data processing system begins executing a new one of the plurality of processes.

4. The data processing means of claim 1 wherein said data processing means includes vector disable means, for disabling said vector processing means without affecting said scalar processing means before said data processing means begins executing a new one of the plurality of processes, and
wherein said vector instruction identifying means includes means for checking said vector processing means to see whether it is disabled.

5. The data processing system of claim 3, further comprising:
scalar processing means for executing scalar instructions in said processes, said scalar processing means including scalar state means for storing scalar state information describing an execution state of said scalar processing means; and
scalar save means for causing said data processing system to store said scalar state information in said memory at one of said portions corresponding to the current process when said data processing system finishes execution of the current process.

6. The data processing system of claim 5, further comprising:
scalar restoring means, responsive to said scalar save means, for causing said data processing system to store into said scalar state means, new scalar state information stored in a portion of said memory corresponding to the one of the new process about to be executed.

7. A method of managing a multitasking data processing system alternately executing portions of a plurality of processes which may contain vector instructions, the one of said plurality of processes currently being executed being called a current process, said data processing system including a separate vector processor to execute said vector instructions simultaneously with the execution of scalar instructions, the method comprising:

a) storing in said vector processor vector state information describing an execution state of said vector processor for a last vector process which most recently executed a vector instruction;

b) continuing to execute scalar instructions until the next vector instruction is to be executed; and

c) storing into memory the vector state information for the last vector process and retrieving the vector state information for said current process into said vector processor if, when the next vector instruction to be executed, the current process is not the same as the last vector process.

8. The method of claim 7 further including the steps of:

a) disabling the vector processor when the last vector process is suspended,

b) taking a vector processor disabled fault when the next vector instruction is to be executed.

c) saving the vector state of the last vector process and restoring the vector state of the current process if the current process is not the same as the last vector process, and

d) enabling the vector processor and retrying the vector instruction causing the vector processor disabled fault.

9. The method of claim 7 wherein said data processing system further includes a scalar processor and stored scalar state information describing an execution state of said scalar processor for said current process, and wherein the method includes the steps of:

a) saving said stored scalar state information in said memory when a new process is started, and

b) restoring saved scalar state information from said memory, said saved scalar state information describing an execution state of said new process.

EPODOC: <EP 0333366A2_1_>

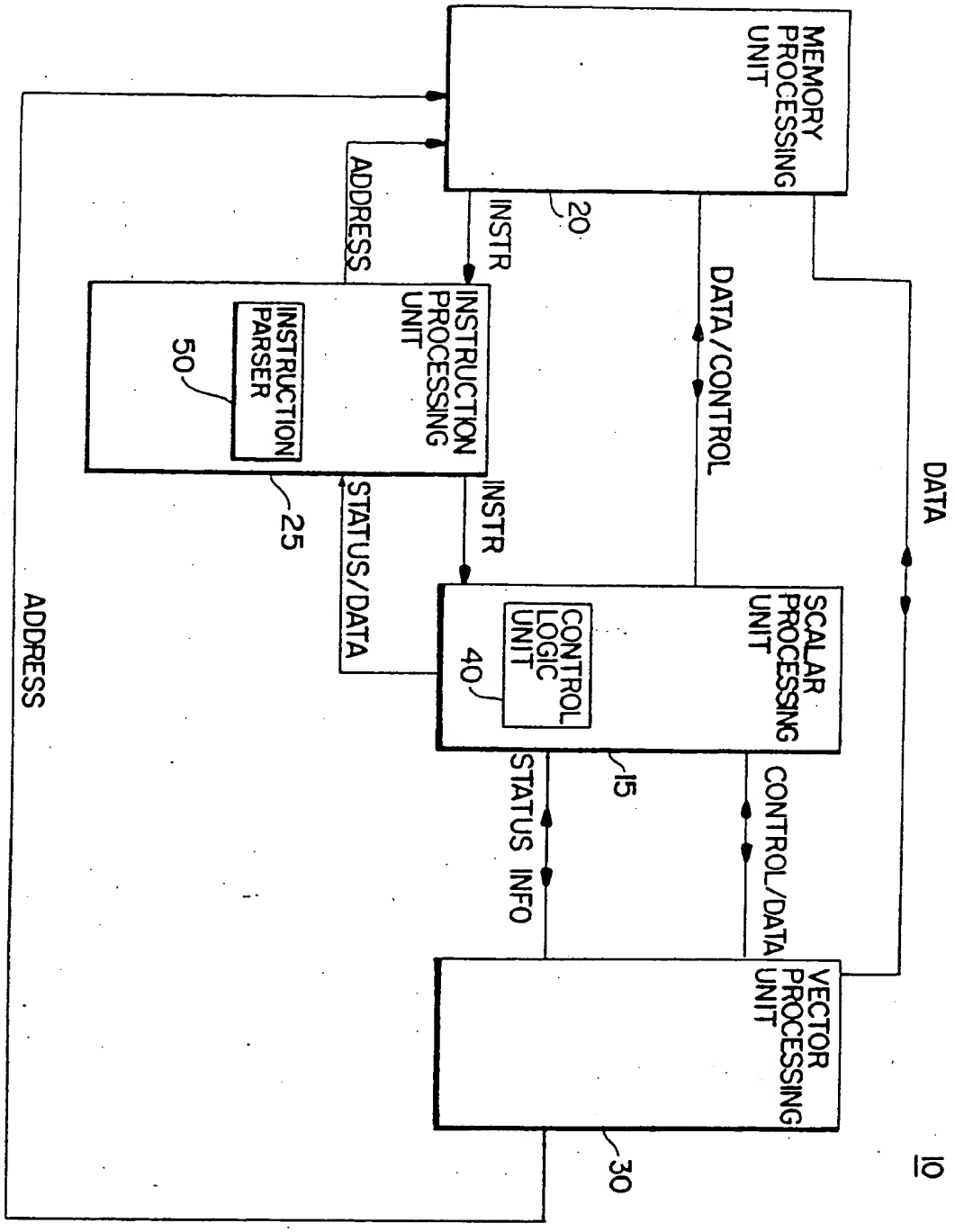


Fig. 1

BNSDOCID: <EP 0333366A2_I_>

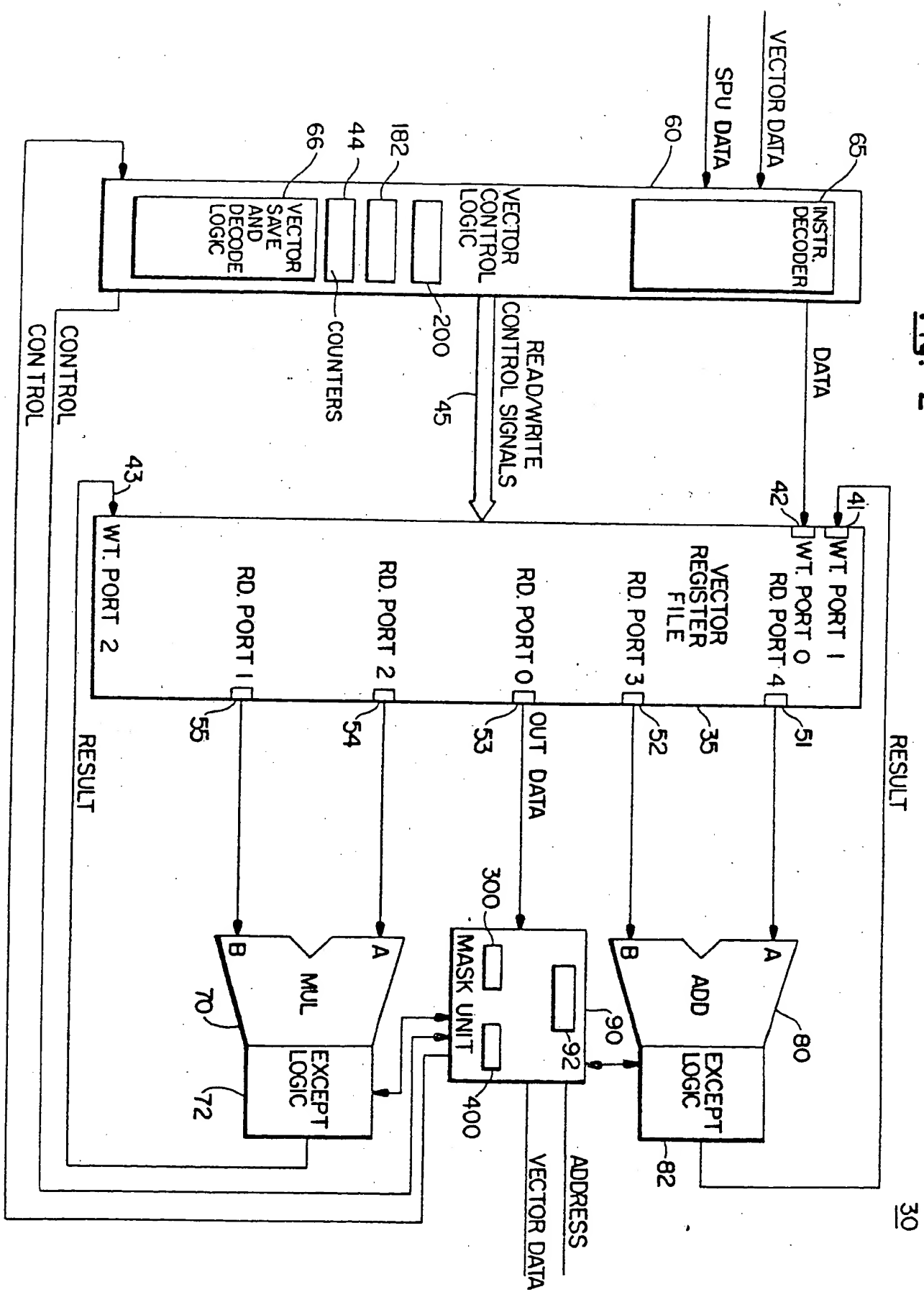


Fig. 2

BNSDOCID: <EP 0333366A2 | >

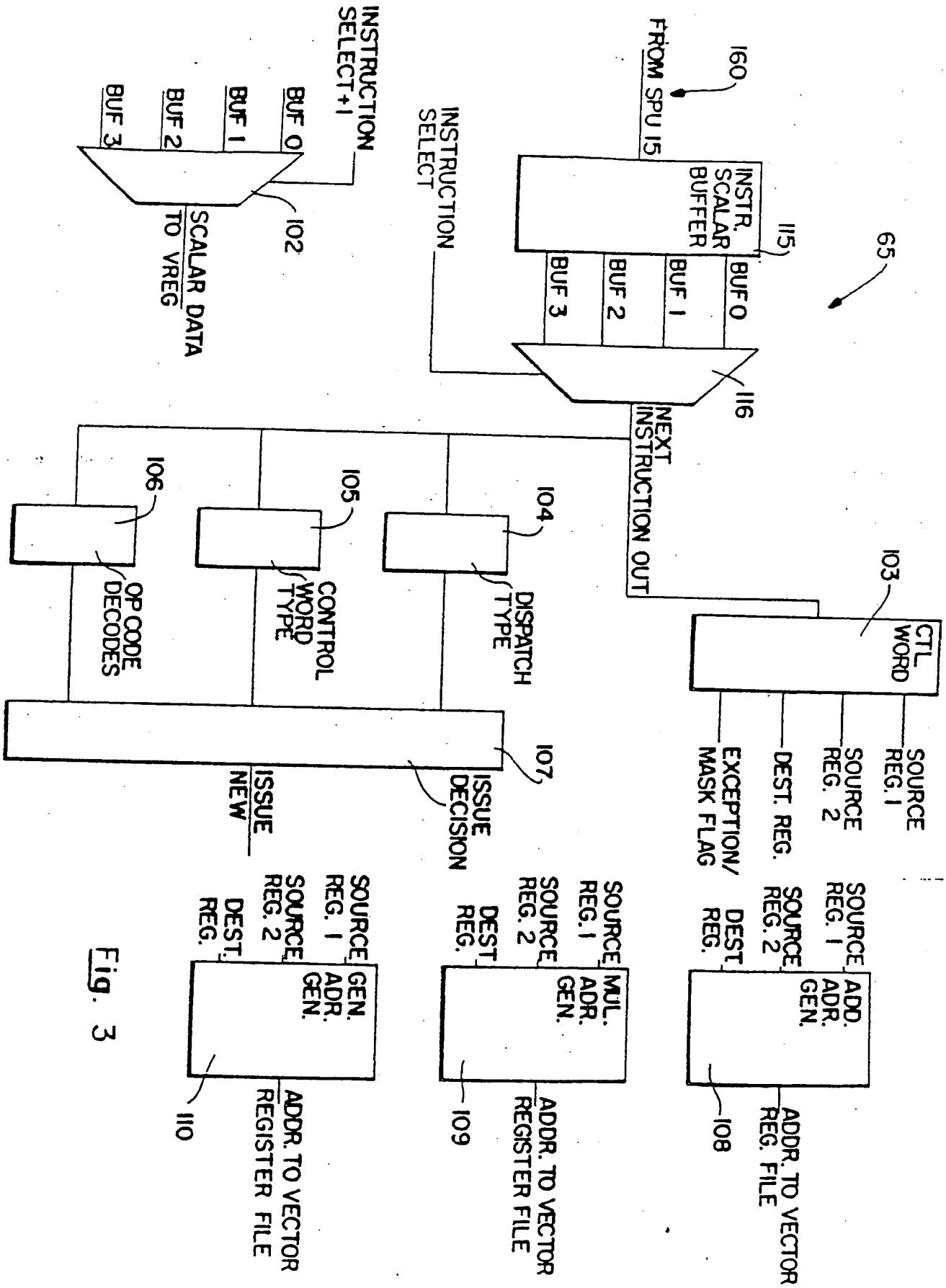


Fig. 3

BNSDOCID: <EP___0333366A2_I_>

Fig. 4a

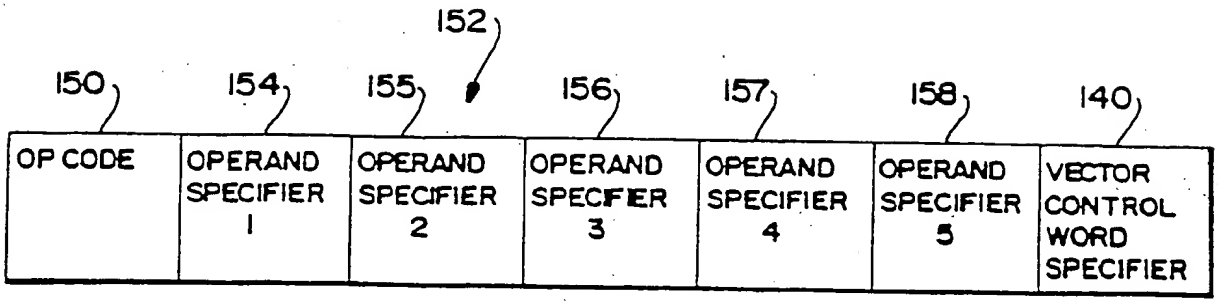


Fig. 4b

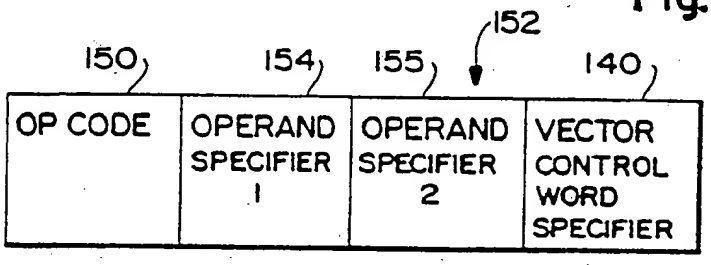


Fig. 4c

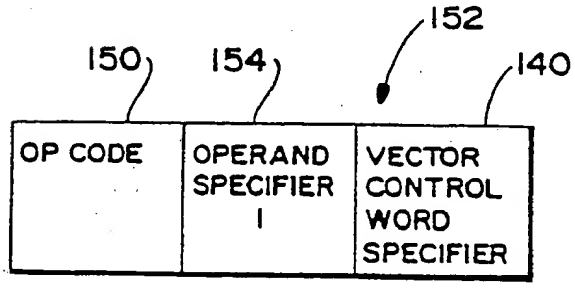
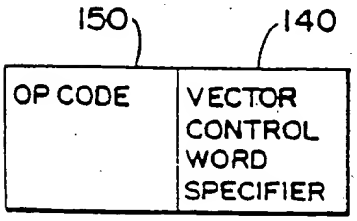


Fig. 4d



BNSDOCID: <EP___0333366A2_I_>

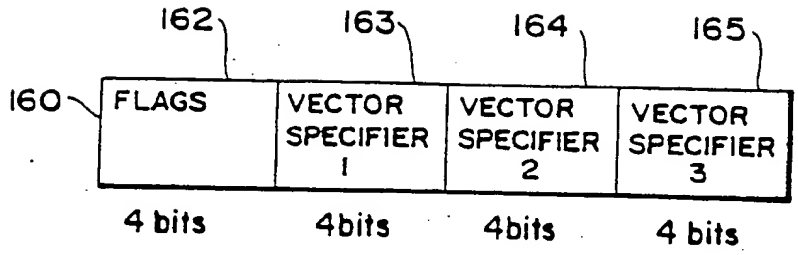


Fig. 5a

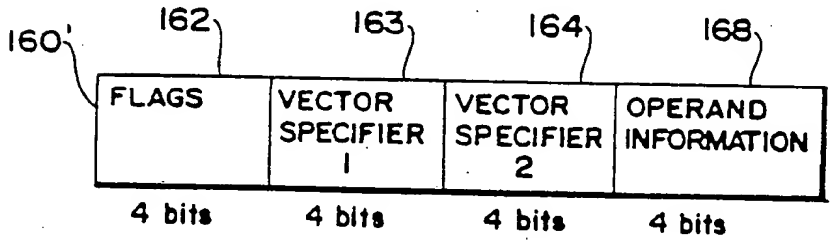


Fig. 5b

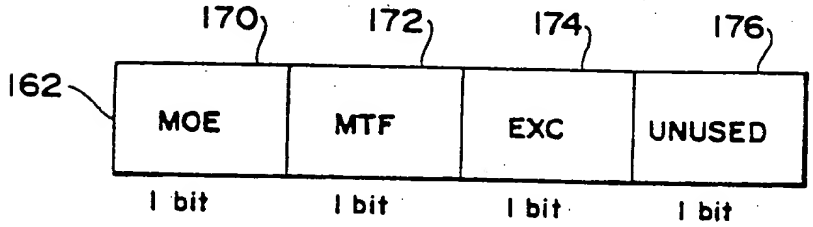


Fig. 6

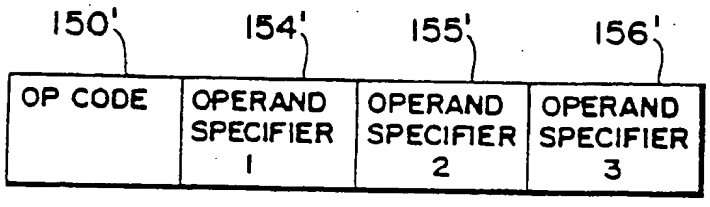


Fig. 7

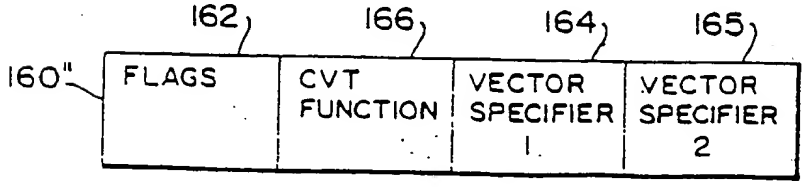


Fig. 5c

BNSDOCID: <EP 0333366A2 I >

Fig. 8a

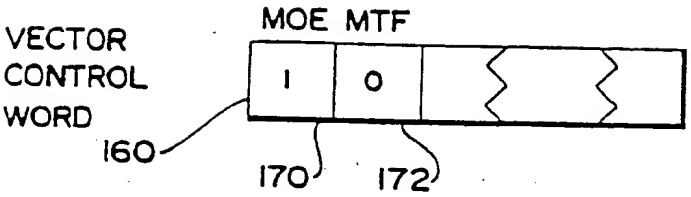


Fig. 8b

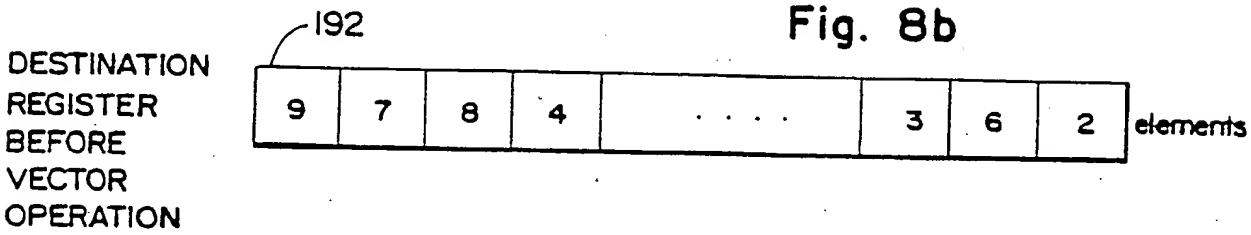


Fig. 8c

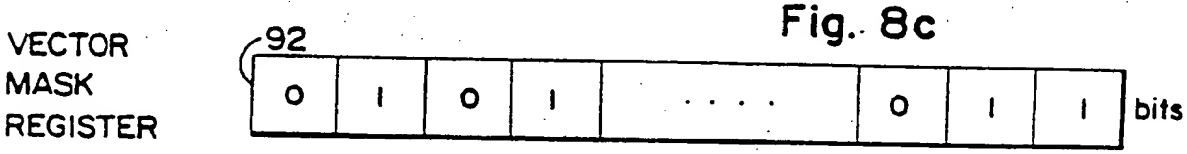


Fig. 8d

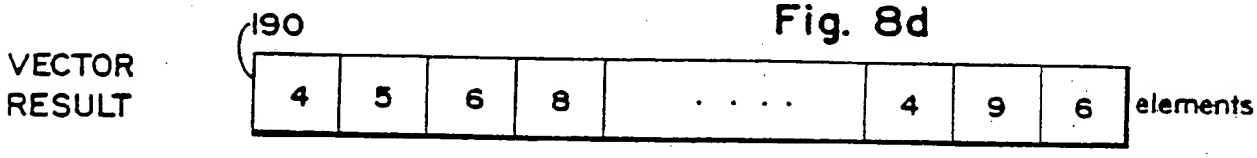


Fig. 8e

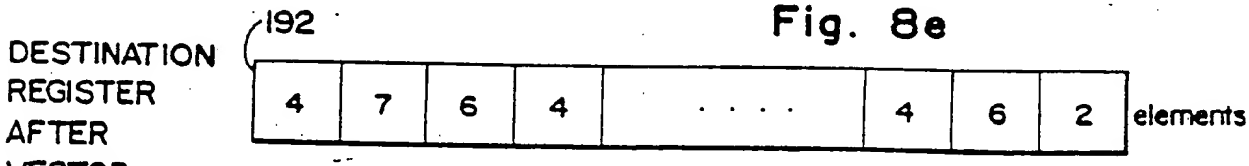
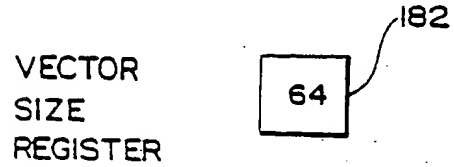


Fig. 8f



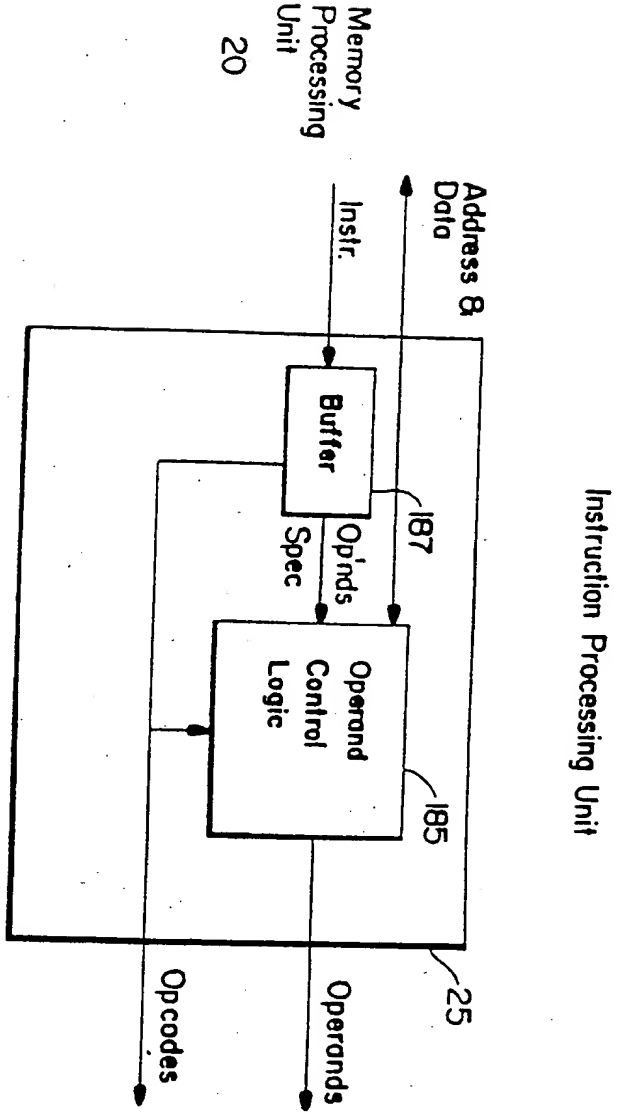


Fig. 9

BNSDOCID: <EP___0333366A2_I_>

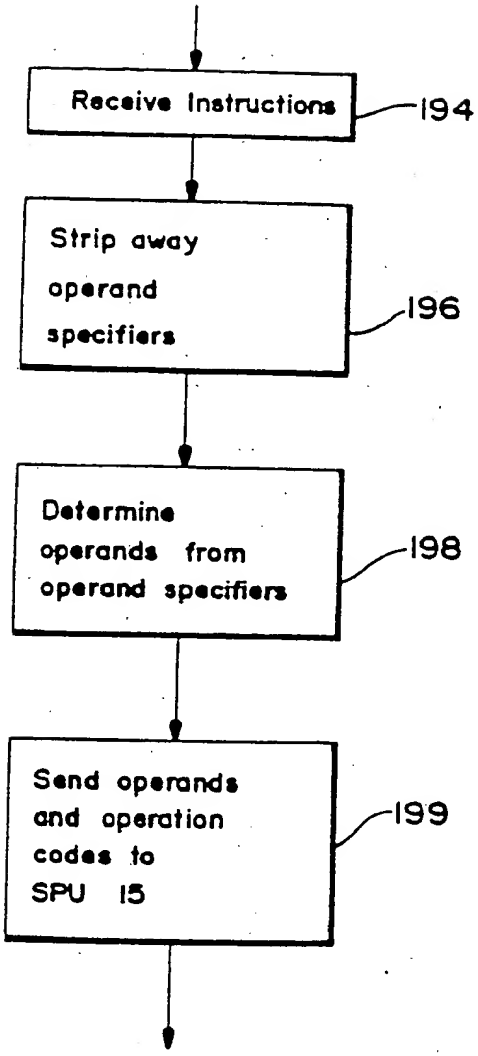
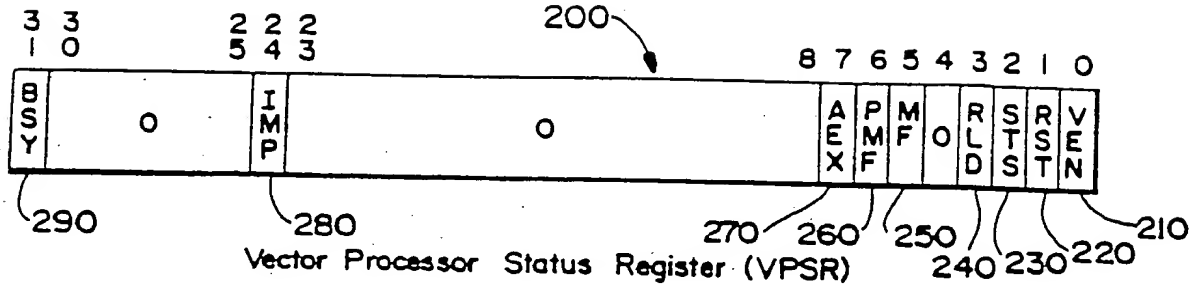


Fig. 10

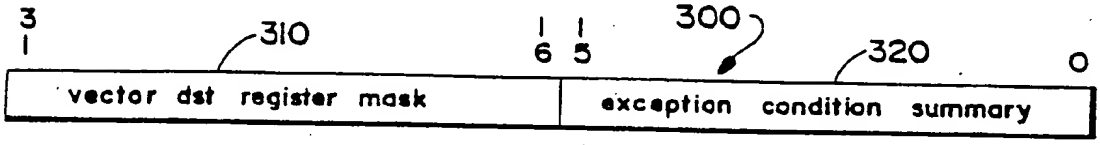
Instruction Preprocessing

BNSDOCID: <EP___0333366A2_I_>



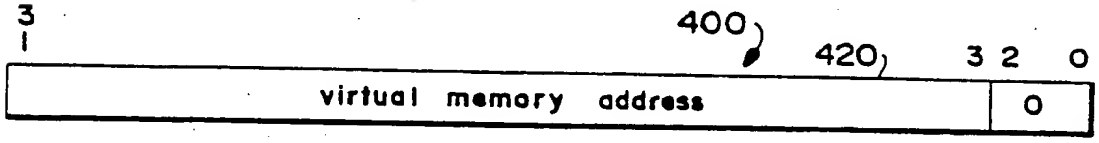
Vector Processor Status Register (VPSR)

Fig. 11



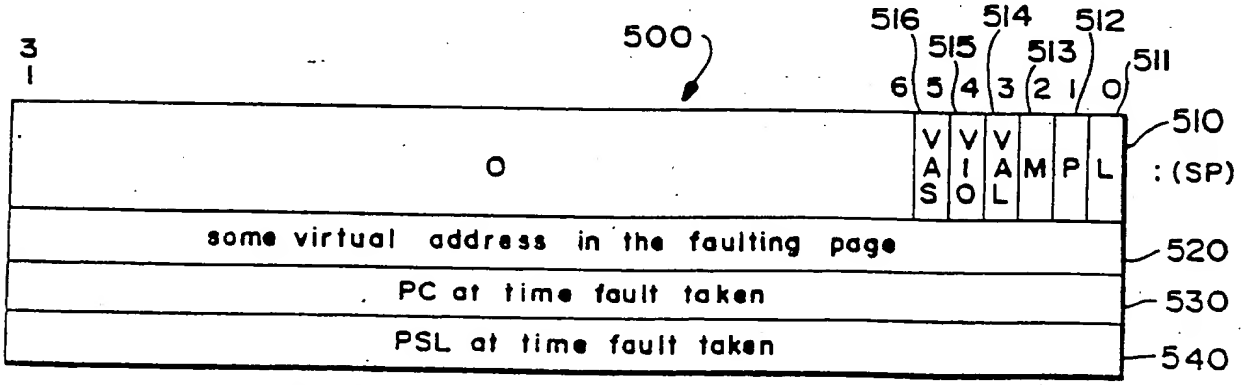
Vector Arithmetic Exception Register (VAER)

Fig. 12



Vector State Address Register (VSAR)

Fig. 13



Memory Management Fault Stack Frame

Fig. 14

BNSDOCID: <EP___0333366A2_I_>

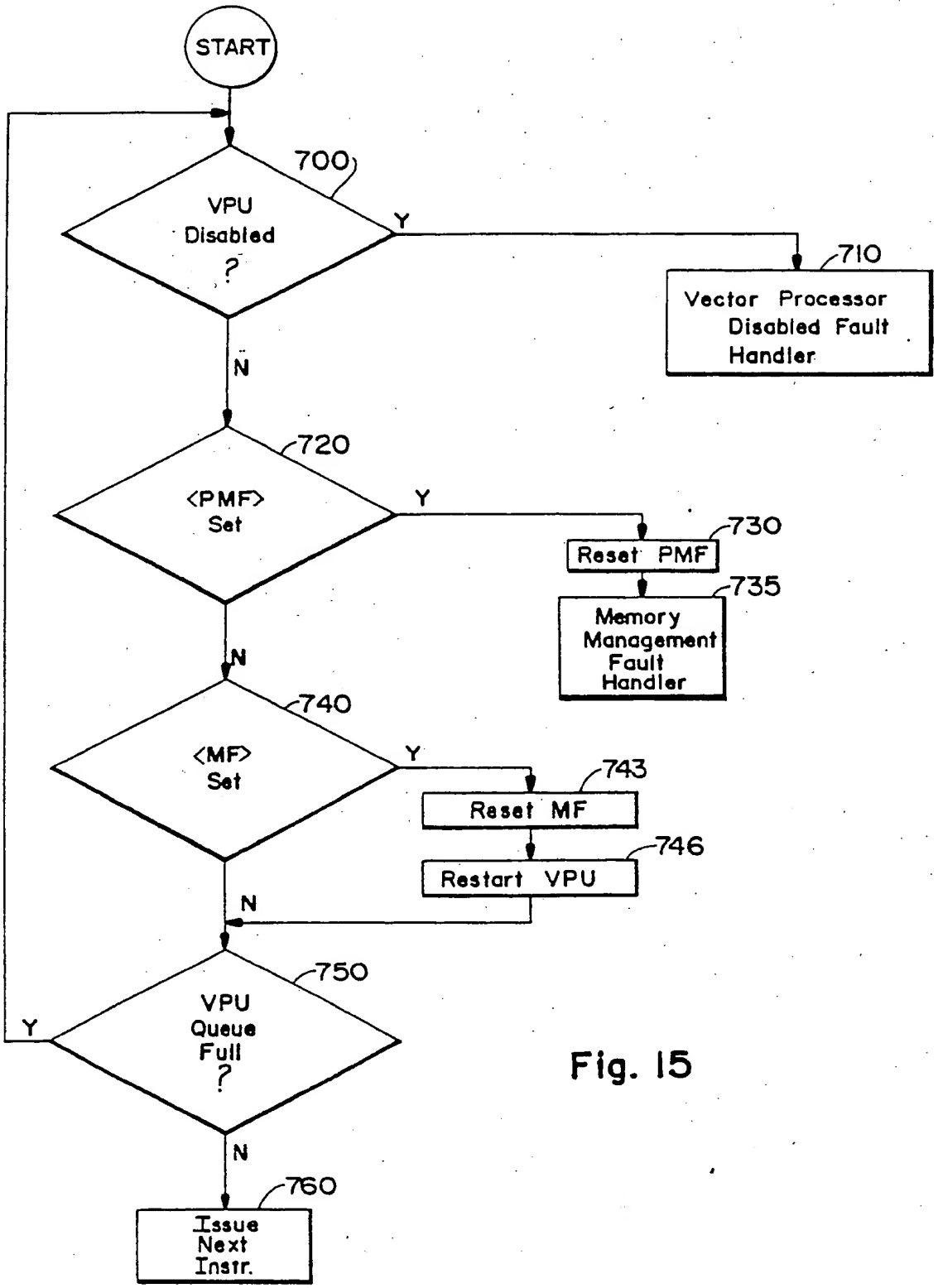
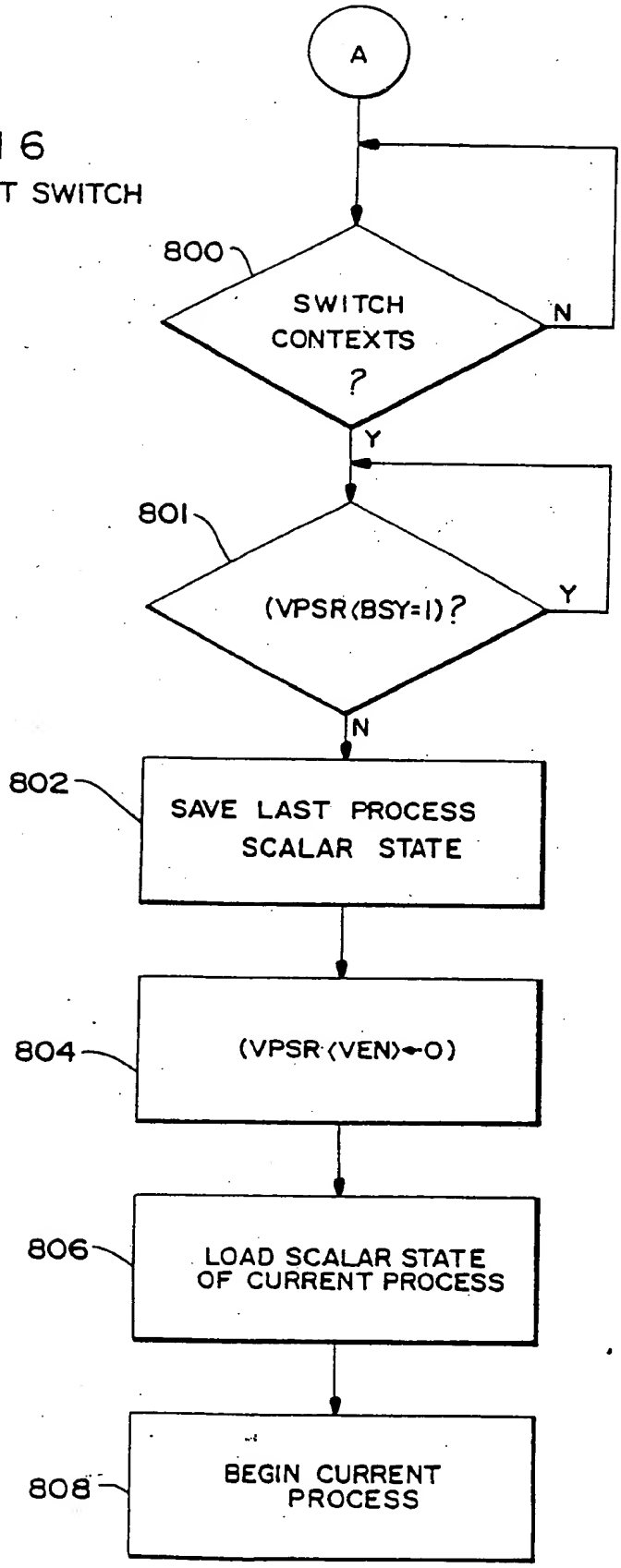


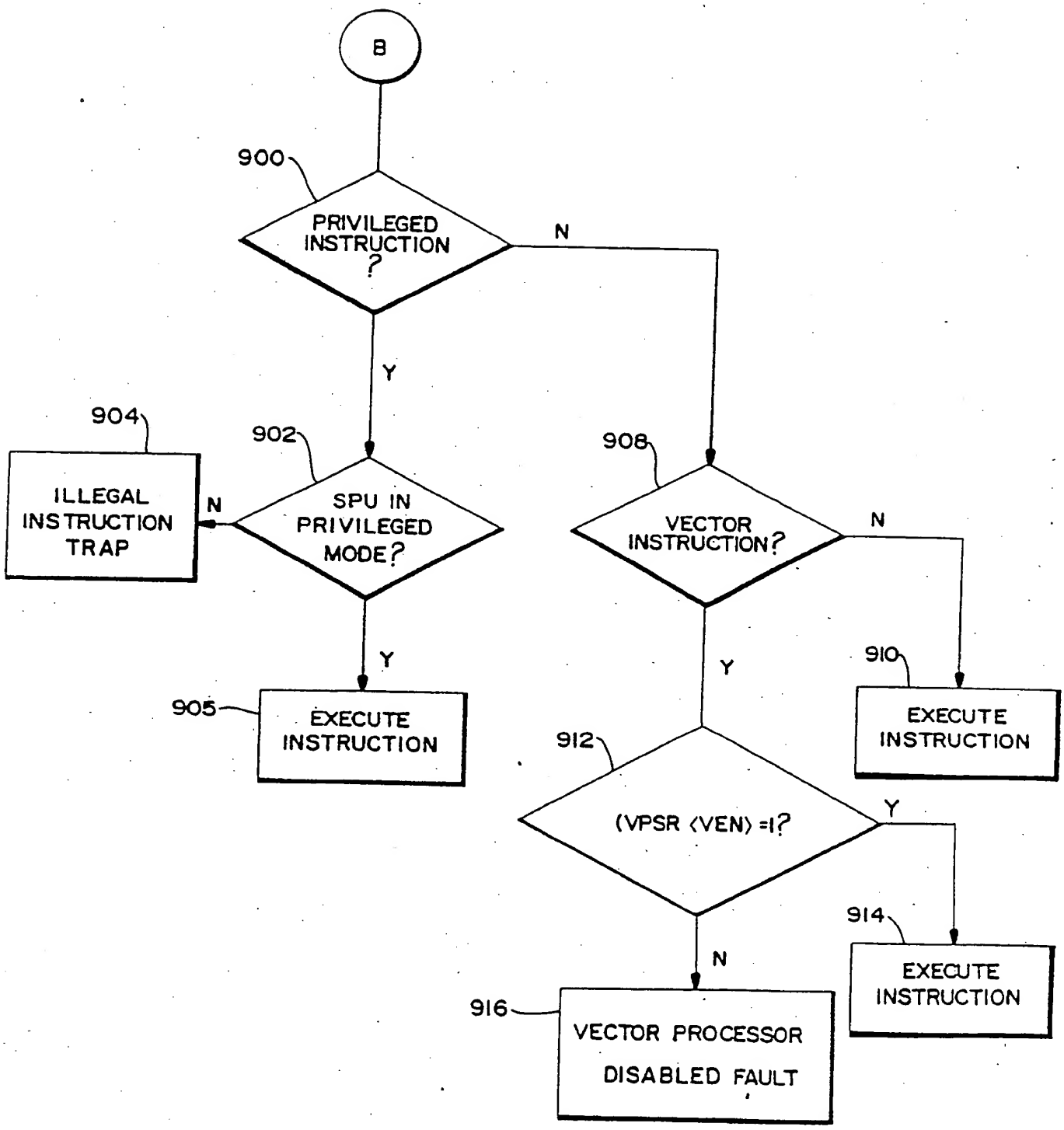
Fig. 15

BNSDOCID: <EP___0333366A2_I_>

Fig. 16
CONTEXT SWITCH



BNSDOCID: <EP___0333366A2_I_>



INSTRUCTION DECODING

Fig. 17

BNSDOCID: <EP___0333366A2_I_>

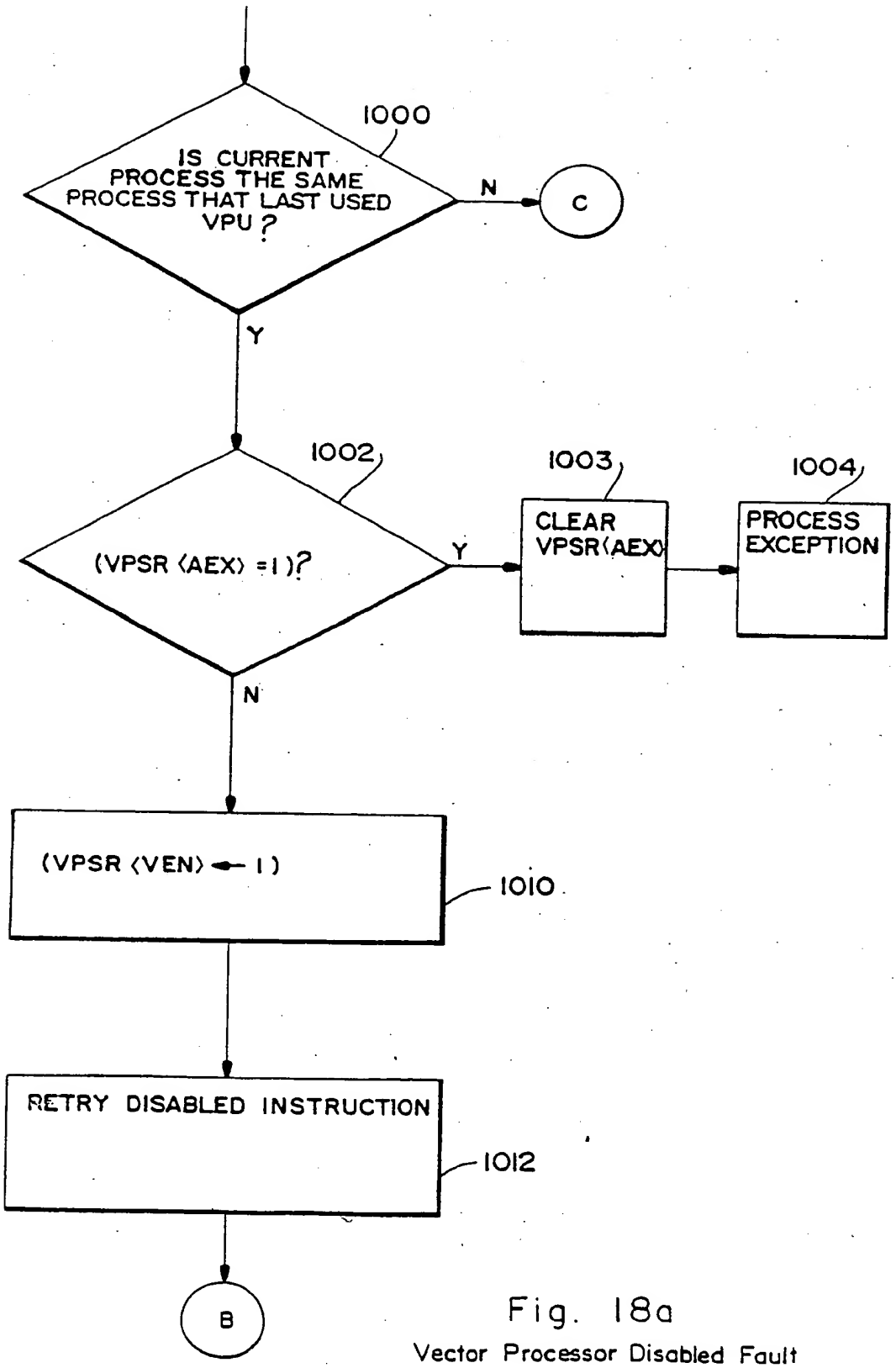


Fig. 18a

Vector Processor Disabled Fault

BNSDOCID: <EP___0333366A2_I_>

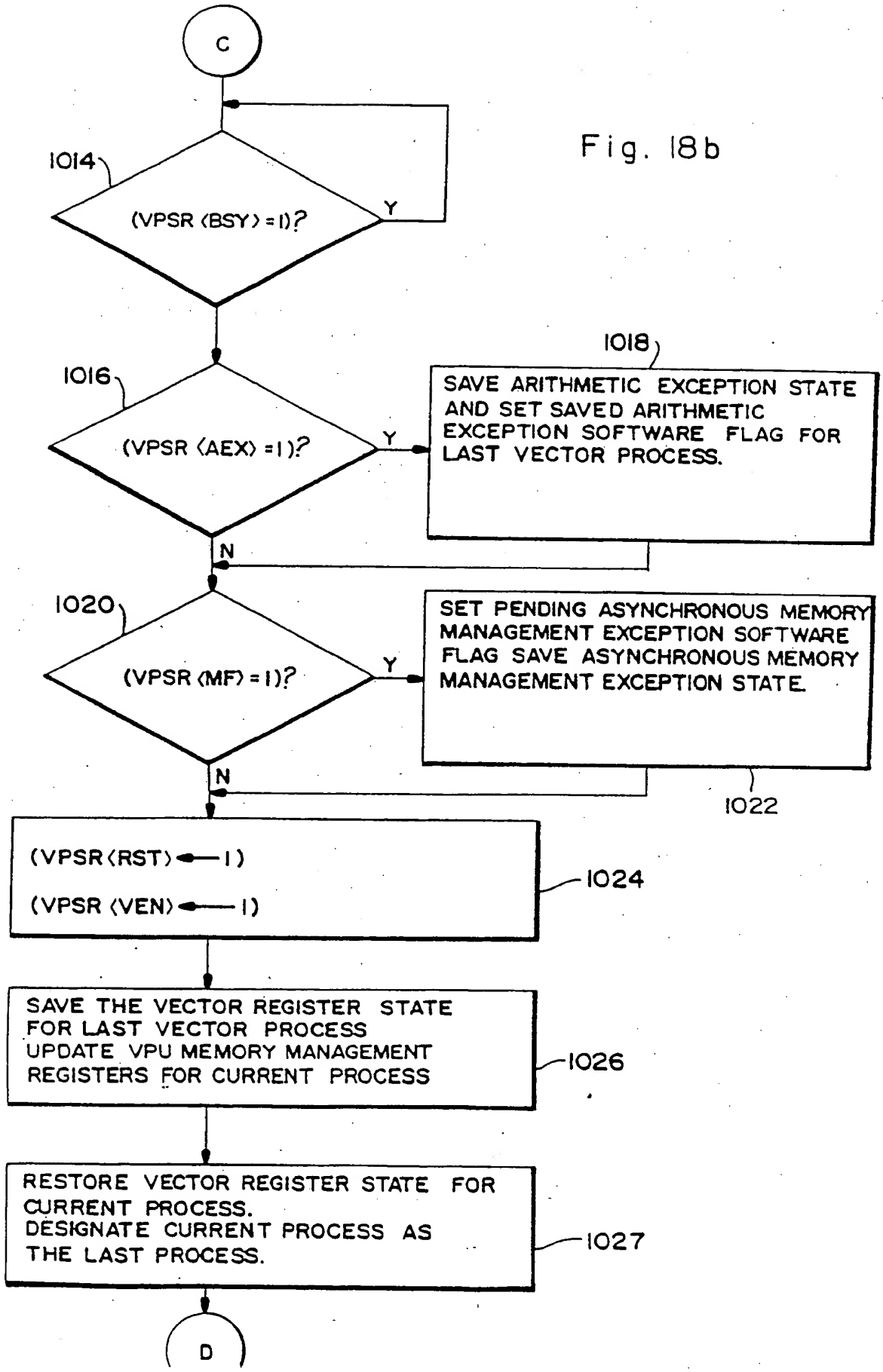


Fig. 18b

BNSDOCID: <EP___0333366A2_I_>

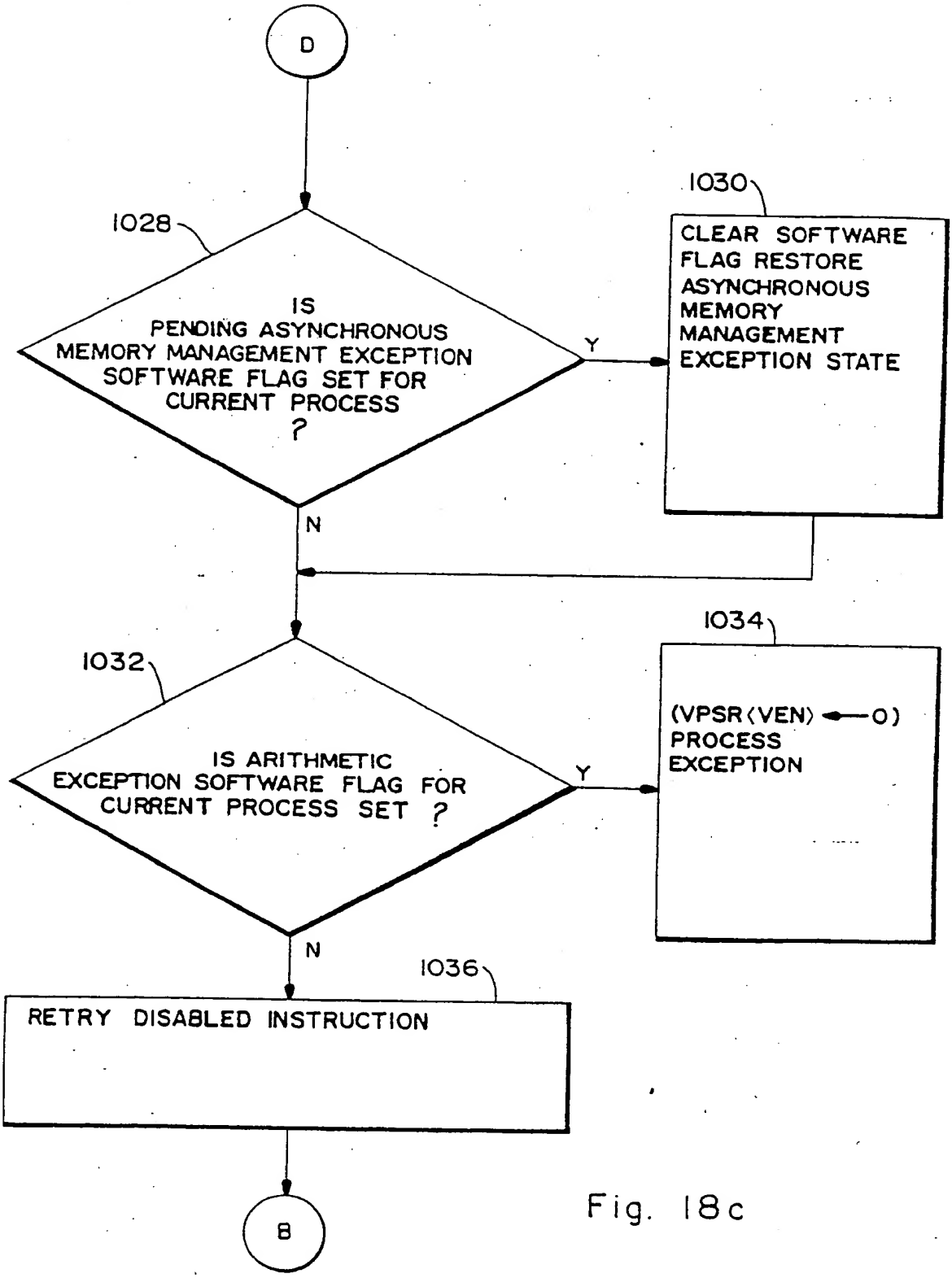


Fig. 18c

BNSDOCID: <EP___0333366A2_I_>

This Page Blank (uspto)

(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) Publication number: **0 333 366 A3**

(12) # EUROPEAN PATENT APPLICATION

(21) Application number: 89302226.9

(51) Int. Cl.[5]: G06F 15/06

(22) Date of filing: 06.03.89

(30) Priority: 18.03.88 US 170399

(43) Date of publication of application:
20.09.89 Bulletin 89/38

(84) Designated Contracting States:
DE FR GB IT

(88) Date of deferred publication of the search report:
02.05.91 Bulletin 91/18

(71) Applicant: DIGITAL EQUIPMENT CORPORATION
146 Main Street
Maynard, MA 01754(US)

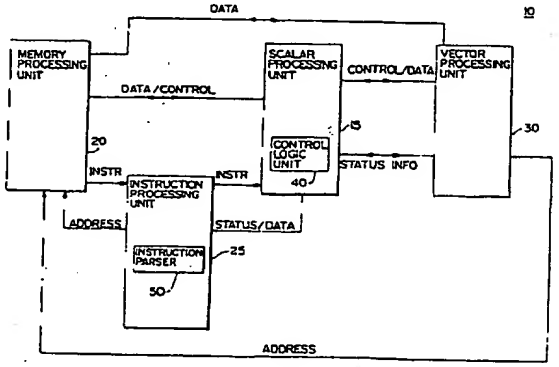
(72) Inventor: Bhandarkar, Dileep P.
3, Lantern Lane,
Shrewsbury Massachusetts 01545(US)
Inventor: Cutler, Dave
3451, Evergreen Point Road
Bellevue WA(US)
Inventor: Cardoza, Wayne
3, Hutchinson Road
Merrimack NH 03054(US)
Inventor: Witek, Rich,
3, Silver Birch
Littleton Massachusetts 01460(US)
Inventor: Orbits, Dave
17028, N.E. 139 Street
Redmond WA 98052(US)

(74) Representative: Goodman, Christopher et al
Eric Potter & Clarkson St. Mary's Court St. Mary's Gate
Nottingham NG1 1LE(GB)

(54) **Context switching method and apparatus for use in a vector processing system.**

(57) The invention relates to a method and apparatus to minimize the time a data processing system spends on saving and restoring vector processor state data during a context switch. A context switch occurs when execution of an old process is suspended and execution of a current process is begun. The vector state data associated with the old process remains in the vector processor until the current process attempts to execute a vector instruction. When this occurs, the vector state data associated with the old process is saved and vector state data associated with the current process is restored, if the old process is not the same as the current process. If the old process is the same as the current process, there is no need to save and restore any vector state data.

Fig. 1



EP 0 333 366 A3

BNSDOCID: <EP___0333366A3_I_>

European Patent Office

EUROPEAN SEARCH REPORT

Application Number

EP 89 30 2226

DOCUMENTS CONSIDERED TO BE RELEVANT

Category	Citation of document with indication, where appropriate, of relevant passages	Relevant to claim	CLASSIFICATION OF THE APPLICATION (Int. Cl. 4)
Y	EP-A-0 239 078 (NEC CORP.) * Abstract; column 3, line 19 - column 4, line 7; column 4, lines 37-48; column 5, lines 39-57; column 6, line 55 - column 7, line 6; column 7, lines 21-54 *	1-3,5-7,9	G 06 F 15/06
A		4,8	

Y	EP-A-0 211 152 (IBM CORP.) * Column 1, lines 25-37; column 2, line 52 - column 3, line 31; column 9, line 53 - column 10, line 3; column 10, lines 16-25 *	1-3,5-7,9	
A		8	

A	EP-A-0 231 526 (HITACHI LTD) * Page 2, lines 1-19; page 3, lines 7-12; page 5, line 18 - page 6, line 21; page 9, lines 15-23; page 13, line 26 - page 14, line 15; page 18, line 25 - page 19, line 12; page 20, line 19 - page 22, line 13 *	1,4,7,8	

TECHNICAL FIELDS SEARCHED (Int. Cl.4)

G 06 F 15
G 06 F 38
G 06 F 46

The present search report has been drawn up for all claims

Place of search	Date of completion of the search	Examiner
THE HAGUE	11-02-1991	WEINBERG L.F.

CATEGORY OF CITED DOCUMENTS

X : particularly relevant if taken alone
Y : particularly relevant if combined with another document of the same category
A : technological background
O : non-written disclosure
P : intermediate document

T : theory or principle underlying the invention
E : earlier patent document, but published on, or after the filing date
D : document cited in the application
L : document cited for other reasons

& : member of the same patent family, corresponding document

EPO FORM 1503 03.82 (P0601)

3NSDOCID: <EP___0333366A3_I_>